Denial of Engineering Operations Attacks in Industrial Control Systems

Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, Vassil Roussev
Department of Computer Science, University of New Orleans
(ssenthiv,sdhunga2,hyoo1)@uno.edu,(irfan,vassil)@cs.uno.edu

ABSTRACT

We present a new type of attack termed *denial of engineering operations* in which an attacker can interfere with the normal cycle of an engineering operation leading to a loss of situational awareness. Specifically, the attacker can deceive the engineering software during attempts to retrieve the ladder logic program from a *programmable logic controller* (PLC) by manipulating the ladder logic on the PLC, such that the software is unable to process it while the PLC continues to execute it successfully. This attack vector can provide sufficient cover for the attacker's actual scenario to play out while the owner tries to understand the problem and reestablish positive operational control. To enable the forensic analysis and, eventually, eliminate the threat, we have developed the first decompiler for ladder logic programs.

Ladder logic is a graphical programming language for PLCs that control physical processes such as power grid, pipelines, and chemical plants; PLCs are a common target of malicious modifications leading to the compromise of the control behavior (and potentially serious consequences). Our decompiler, *Laddis*, transforms a low-level representation to its corresponding high-level original representation comprising of graphical symbols and connections. The evaluation of the accuracy of the decompiler on the program of varying complexity demonstrates perfect reconstruction of the original program. We present three new attack scenarios on PLC-deployed ladder logic and demonstrate the effectiveness of the decompiler on these scenarios.

CCS CONCEPTS

• **Security and privacy → Denial-of-service attacks**;

KEYWORDS

Disassembler, Ladder logic, PLC, SCADA, Industrial Control System, Forensics, Protocol Reverse Engineering

ACM Reference Format:
Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, Vassil Roussev. 2018. Denial of Engineering Operations Attacks in Industrial Control Systems. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3176258.3176319

CODASPY '18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5632-9/18/03…$15.00
https://doi.org/10.1145/3176258.3176319

1 INTRODUCTION

Programmable logic controllers (PLCs) are embedded devices used in industrial control systems (ICS) to automate the control and monitoring of physical industrial and infrastructure processes such as gas pipelines, nuclear plants, power grids, and wastewater treatment facilities [11]. Thus, their safety, durability, and predictable response times are the primary design concerns. Unfortunately, they are not designed to be resilient against cyberattacks [10]. If a PLC is compromised, the physical process controlled by the PLC is also compromised which could lead to a catastrophic incident.

To compromise a PLC, an attacker infiltrates into an ICS network to communicate with the PLC, or gains physical access of the PLC to communicate using local USB or serial ports. The attacker may attempt to modify either the firmware (or the operating system) of the PLC, or the control logic (typically written in the languages defined by `IEC 61131-3` such as Ladder Logic). The firmware is hard to modify for the attacker because it has to be signed by the corresponding vendor's private key [19]. The attacker may find and utilize any debugging/testing ports (such as JTAG/UART) at hardware level to compromise the firmware [17]. However, this requires physical access, which makes it impractical as a remote attack.

The control logic defines how a PLC controls a physical process. Unfortunately, it is vulnerable to malicious modifications because PLCs either do not support digital signatures for control logic, or the ICS operators do not use/configure them. The adversaries target the control logic to manipulate the control behavior of the PLC. For instance, Stuxnet targets Siemens `S7-300` PLCs that are specifically connected with variable frequency drives [15]. 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). The attack involves disrupting the normal operation of the motors by changing their motor speed periodically from 1,410 Hz to 2 Hz to 1,064 Hz and then over again.

For the purposes of our discussion, we define *engineering operations* as a continuous cycle of developing and updating the PLC control logic in response to changing operational requirements in ICS. A vendor-supplied programing software is used to create a control logic program and then to transfer it to/from a remote PLC over the network. In case of an incident, a forensic investigator (or a control operator) is likely to use the software to acquire the control logic from a suspicious PLC since the PLCs are often located at remote sites that may be difficult to get to.

We present three new attack scenarios, referred to as *denial of engineering operations* (DEO) attacks that subvert the capability of the programing software to acquire the actual control logic from an infected PLC. The first two attacks employ man-in-the-middle approach to control the network traffic during attempts to acquire

the control logic from an infected PLC. In the first (and most conventional) attack, the attacker removes the infected code from the packets to hide the infection, such that the programing software shows the original (uninfected) logic to the investigator. In the second attack, the attacker replaces the selected control logic instructions in the packets with noise; in our experiments, this caused confusion and crashed the software. In the third attack, the attacker creates a well-crafted malicious control logic that runs on a PLC successfully but crashes the software when attempting to acquire the control logic from the PLC. This attack does not employ man-in the middle and allows the attacker to leave the network after transferring the malicious control logic to the PLC.

Since the firmware is intact in these attacks, the PLC sends the entire (infected) control logic to the programing software. Although the attacker intercepts the traffic and/or crashes the software, the original (infected) control logic can be captured and extracted from the network traffic for forensic analysis. However, the control logic being transferred is the compiled version, which is the binary (low-level representation of) control logic.

This paper presents a decompiler `Laddis` for the ladder logic, which is a widely-used programing language for PLCs. `Laddis` is the first decompiler developed for ladder logic programs. It supports ladder logic instructions extensively. `Laddis` extracts and decompiles a ladder logic program from the network traffic on both directions i.e. when a ladder logic is downloaded to and uploaded from a PLC. The programing software downloads a ladder logic program to a PLC to configure and/or update the logic in the PLC. However, it cannot show a ladder logic program from the download stream of an attacker. The only option is to let the attacker download the program completely to a target PLC, and then use the software to upload the program from the PLC, which is unacceptable since it may damage the physical process being controlled by the PLC.

For the evaluation, we use Allen-Bradley Micrologix 1400-B PLC and RSLogix 500 programing software. `Laddis` is currently developed and tested for RSLogix 500. We created 91 ladder logic programs of varying complexity and also downloaded 58 programs from PLCS.net [7] to evaluate the accuracy of `Laddis`. The downloaded programs are developed by different programmers for different physical processes such as traffic lights, an elevator, and a train crossing. We further use six versions of RSLogix 500 to generate six instances of each ladder logic program. Our evaluation results show that `Laddis` is compatible with all the versions of RSLogix and decompiles all the binary programs with 100% accuracy. Furthermore, `Laddis` was evaluated against the proposed three attacks and decompiles the infected programs successfully.

The contribution of this work is threefold: a) we identify new attack scenarios specific to the manner in which ICS are operated; b) we present a new decompiler tool that can reconstruct the attack code from network capture; and c) we show how to perform forensic analysis in response to the attacks

2 NEW ATTACK SCENARIOS

We start our discussion with an outline of the three new attack scenarios that incapacitate PLC programming software by limiting its abilities of acquiring and displaying a ladder logic program from a PLC. We briefly introduce the three attack scenario in this section, and we will describe them in detail in section 6.4 along with the usage of `Laddis`.

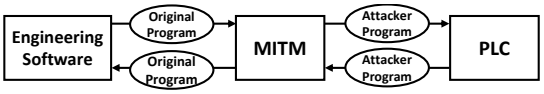


Figure 1: DEO Attack 1 – Hiding infected ladder logic (running in a target PLC) from the engineering software.

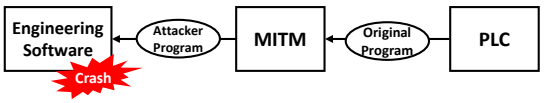


Figure 2: DEO Attack 2 – Crashing the engineering software.

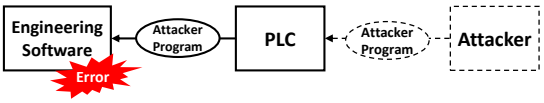
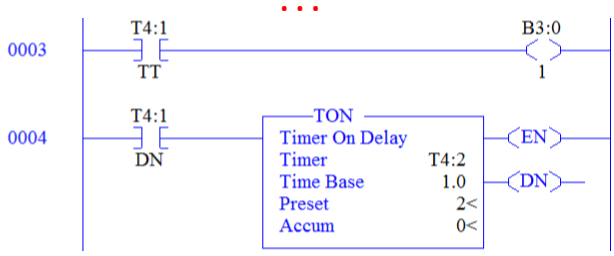


Figure 3: DEO Attack 3 – Injecting a crafted ladder logic program to a PLC that crashes the engineering software.

DEO Attack 1: Hiding infected ladder logic (running in a target PLC) from the PLC programing software. In this attack scenario, an attacker performs man-in-the-middle between a PLC and an engineering workstation running a PLC programing software. When a ladder-logic program is being downloaded from the programming software to the PLC, the attacker replaces a part of an original ladder logic with an infected logic. When a control engineer tries to upload the (infected) program from the PLC to the programing software, the attacker intercepts the traffic and replaces infected part with the original logic. Hence, the software displays the normal program that clearly deceives the control engineer (Figure 1).

DEO Attack 2: Crashing the PLC Programing Software. In this attack scenario, the attacker employs the man-in-the-middle to intercept the network traffic between the programing software running on an engineering workstation and a target PLC. When a ladder logic program is being uploaded from a PLC to the programing software, the attacker intercepts the traffic and changes the original ladder logic programs into malformed programs by replacing ladder logic instructions with noise such as the sequence of `0xFF bytes`. The software crashes when it tries to process the malformed ladder logic program (Figure 2).

DEO Attack 3: Injecting a crafted ladder logic program to a PLC that crashes the PLC programing software. This attack scenario is the stealthiest among the three proposed scenarios. The attacker creates a well-crafted (malicious) ladder-logic program at the binary-level that runs on a target PLC successfully. However, when the programing software tries to acquire the program from the PLC, it cannot process it and gives an error, causing the denial of service (refer to Figure 3). This scenario does not involve any man-in the middle attacks and allows the attacker to leave the ICS network after installing the malicious program to the PLC.



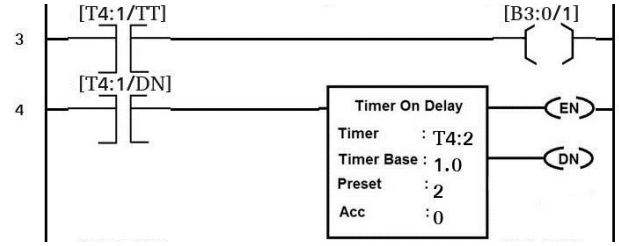
a) Ladder-logic source code snippet of a traffic-light program

```
 78 00 00 4A 2E 16 00 E4 00 0A 04 D4 4F 0E   J.  ‰   'O
 91 00 BC 00 01 03 CC 4F 01 00 00 00 58 B4  °   ÃO     X¥
104 18 00 E4 00 0A 04 D4 4F 0D 00 58 01 0A   ‰   'O  X
117 00 96 04 DA 4F 02 00 00 00 3E C7 16 00  ñ ⁄O     >«
```

b) Binary ladder-logic snippet of a traffic-light program

```
Rung-3: (XIC/[T4:1/TT]) --> (OTE/[B3:0/1])
Rung-4: (XIC/[T4:1/DN]) --> (TON/[T4:2/1.0/2/0])
```

c) *Laddis* ASCII output of decompiling the binary ladder-logic snippet



d) *Laddis* graphical output of decompiling the binary ladder-logic snippet

Figure 4: Overview of the Decompiler `Laddis`. A ladder logic program is compiled to a binary ladder logic, which is then decompiled by `Laddis`

3 OVERVIEW OF THE DECOMPILER - LADDIS

Figure 4 presents the three stages of a ladder logic program, beginning from the creation of the program (source code) (Figure 4(a)), which is then compiled into a binary form that is run by a PLC (Figure 4(b)). Finally, the proposed decompiler `Laddis` transforms the binary ladder logic back to its original form i.e. source code (Figures 4(c) and 4(d)).

Ladder-logic Program. Ladder logic is widely-used to program PLCs. It is a graphical language comprising of symbols that are placed together to form a control logic. Each symbol is an instruction such as timer, counter, input, output, move etc., and is analogous to opcode in Assembly language. It works on data to affect the current state of a physical process. The data is referenced in the symbols and is analogous to an operand.

A ladder logic program is divided into rungs (horizontal lines) placed one after another (similar to the steps of a ladder). Each rung has symbols that can be placed in series or parallel depicting `AND` or `OR` logic respectively. The execution of the program starts from the instructions of the first rung from left to right and then moves to the next rung in the sequence.

Figure 4(a) depicts a ladder logic code snippet of a traffic-light program that controls red, orange and green lights using timers. The snippet shows the third and the fourth rungs of the program, each has two instructions (or symbols) placed in a series. When a timer runs, it turns on its respective light; after the timer completes its execution, it turns off the light.

Binary ladder logic. A PLC does not run graphical symbols of ladder logic directly. A ladder logic program is compiled into its binary (low-level representation) form that a PLC can execute. Figure 4(b) depicts the code snippet of the binary ladder logic of the traffic-light program, which contains rungs and instructions at binary-level. (The next section presents the details)

Decompiler. We develop `Laddis` that understands the binary ladder logic and decompiles it into its high-level representation in a human-readable form. Figures 4(c) and 4(d) present the snippet of a decompiled code generated by `Laddis`. It is the traffic-light program (shown in Figure 4(a)). The `Laddis` output clearly identifies rungs and the logic in each rung. In Figure 4(c), the output only consists of the ASCII characters that is useful for automating further forensic analysis such as via Python scripts. `Laddis` replaces the graphical symbols with their equivalent instruction codes derived from the abbreviations of the instructions. For instance, TON is used for the `Timer-On-Delay` instruction, XIC for `Examine-If-Closed`, and OTE for `Output-Energizer`. In Figure 4(d), `Laddis` takes its ASCII output as input and generates the graphical ladder logic.

The ladder logic instructions contain data or the addresses of the data. For instance, timer instruction has three types of data i.e. Timer Base, Preset, and Accum. The OTE has the address of the bit representing the current state of a traffic light i.e. ON or OFF. The `Laddis` output (Figures 4(c) and 4(d)) is complete and equivalent of the original code (in Figure 4(a)), which includes rungs, instructions and their respective data and the addresses of the data.

4 LADDIS DECOMPILATION INTERNALS

To decompile a binary ladder logic, a number of challenges are involved including understanding the anatomy of rungs and different ladder logic instructions at binary-level. Apparently, the binary ladder logic file is not self-contained for an accurate decompilation and the associated data and configurations are required.

It is worth mentioning that we use Allen-Bradley's RSLogix 500 programing software to discover the internals of a binary ladder logic. The findings may reflect RSLogix 500 internals. However, it does not hinder the goal of this work, which is to evaluate the effectiveness of this decompiler under the three proposed attacks. These attacks subvert the capability of a programming software to acquire a ladder logic program from a remote compromised PLC, restricting a forensic investigator from acquiring and analyzing an important piece of evidence.

This section further discusses the internals of a binary ladder logic and how they are used by `Laddis` to perform decompilation.

4.1 Identifying the Rungs

Figure 5 shows an example of a binary ladder-logic with the labels for the bytes of the first rung. A binary ladder-logic starts with a rung and may consist of multiple rungs located in a sequence. Each rung always starts with two bytes of zero values, followed by two bytes containing a signature of the rung. If a rung has multiple exact instances in the binary logic, all the instances have the same signature. The fifth and sixth bytes are the rung size. The instructions start from the sevenths bytes (discussed in the next section). The size of the rungs varies depending on the type and the number of the instructions.

`Laddis` uses the rung-size field to identify the rungs in a binary logic. Since the rungs are located contiguously in a sequence, `Laddis` starts with the first rung and then traverses the subsequent rungs; each rung always starts with `0x00` and `0x00`.

4.2 Identifying the Instructions in a Rung

The instructions in a rung are located contiguously in a sequence, starting from the seventh byte of each rung. An instruction comprises of an opcode (two bytes), a file-number (one byte), an operand consisting an offset of a word-address (two bytes) and a bit address (two bytes). The *Opcode* represents the operation of the instruction such as `XIC` (Examine-if-closed) and `RES` (Reset). It typically works on data; each type of data has an assigned number that is referred to as file-number. For instance, 0x00, 0x01, 0x03 are used for output, input and status data respectively. The operand points to the data of interest. The operand size may vary in the instructions, thereby varying the length of the instructions. For instance, the instructions `JMP` (Jump-to-label), `SUB` (Subtraction) and `ASC` (ASCII-String-Search) are 10, 22 and 28 bytes long. Some instructions such as `END` do not have an operand (the last rung in the last line in Figure 5).

Branch instruction is an exception. It is used to place the symbols in parallel in a ladder logic source code. At the binary-level, branch instruction does not have the prescribed structure of an instruction and comprises of three components, 1) branch start, 2) branch continue, and 3) branch end; each component has a specific unique code at binary-level (similar to opcode) i.e. `0x0800D402`, `0xD0021000`, and `0x14000C00` and `0xD0020C00` respectively.

To identify each instruction in a rung, `Laddis` maintains the mapping between ladder logic instructions (the graphical symbols) and their binary-representation along with the size of the instructions and their short forms in ASCII characters. Table 1 lists the type of ladder logic instructions that are supported by `Laddis`. The total supported instructions are 120 (refer to Table 8 in appendix for the complete list).

4.3 Obtaining the Addresses in the Instructions

The instructions use addresses to point to the data of interest. Figure 8 presents an example of the addressing format i.e. `O:1/3`. '0' refers to the type of file/data i.e. output. After colon, '1' and '3' are the *word* and *bit* index numbers that are pointing to a specific bit.

The instructions at the binary level contain the *bit index number* and the *file number* (pointing to the type of a file/data). However, it does not have the *word index*. Recall that a binary ladder logic is not self-contained to perform accurate decompilation. In this

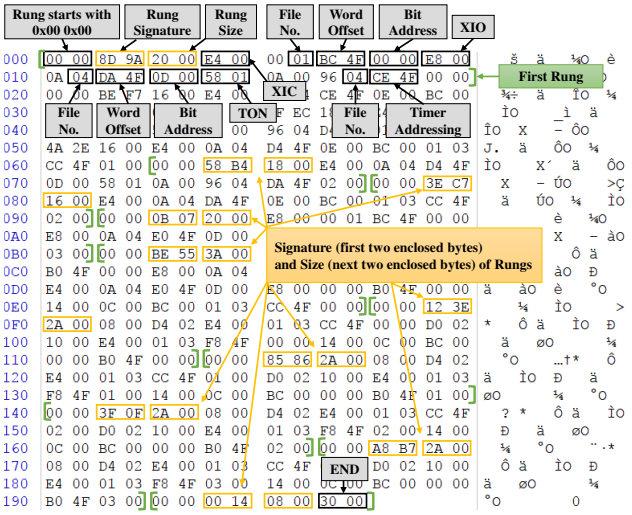


Figure 5: Binary ladder logic consisting of several rungs; each has multiple instructions. A pair of brackets encloses a rung.

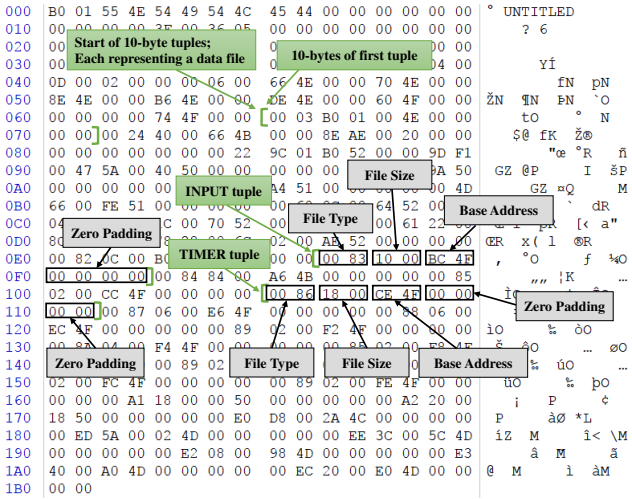


Figure 6: Configuration file for addressing in the instructions.

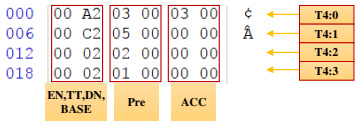


Figure 7: Timer file for the data in the timer instruction.

case, it requires the associated configuration file to determine the *word index*. Figure 6 shows the configuration data. It contains 10 byte-tuple for each type of file/data t (such as timer, input, output etc). Each tuple has a base-address (let say, B_t). In binary ladder logic, an instruction contains a *word offset* (let say, O_t). To find out the *word index* for an instruction, $O_t - B_t$ is performed. For instance, in Figure 5, the first instruction in the first rung is XIC.

Table 1: Ladder logic instructions supported by `Laddis`.

Instruction Type	No. of Instructions	Minimum Inst. Size	Maximum Inst. Size
Bit	9	8	16
Timer	14	2	16
Input/Output	6	2	22
Compare	8	16	22
Compute	10	16	22
Move/Logical	7	10	22
File/Misc	16	10	46
File Shift/ Sequences	9	22	28
Program Control	8	2	10
ASCII Control	8	16	28
ASCII String	6	16	28
High Speed Counter	2	16	34
Trigonometry	6	20	20
Advanced Math	11	16	40
TOTAL	120	-	-

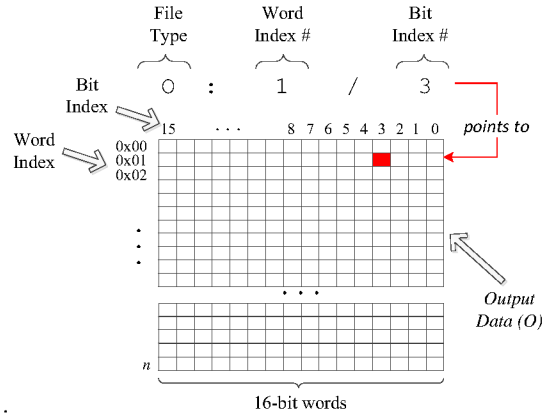


Figure 8: Addressing in the instructions. `O:1/3` address is used for illustration.

Its word offset is `0x4FCE`. In the configuration file, the base-address of the input file (pointed to by the `XIC`) is `0x4FBC`. Hence, the *word index* of the `XIC` is `0x12`.

`Laddis` parses the configuration file, obtains the relevant base addresses for the instructions in a binary ladder logic and then, resolves the addresses by subtracting the *word offsets* with the *base addresses*.

4.4 Obtaining the Data in the Instructions

Some instructions (such as timer and counter) have parametric data that is not present in binary ladder logic. For instance, a timer instruction has three parameters, `Base`, `Preset` (Pre) and `Accumulated` (ACC). The *Base* sets the interval of timing. When timer runs, it keeps incrementing ACC until it reaches to the *Preset* value, which resets the timer. These parameters are stored in a separate timer file (Figure 7). Each parameter is of two bytes and appears in the following sequence: `Base`, `Pre` and `ACC`.

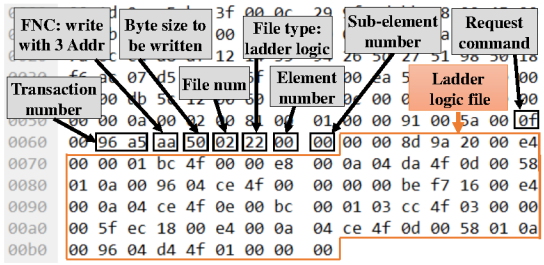


Figure 9: File extraction from downloading traffic

A ladder logic program may define and use multiple timers. To determine the correct parameters associated with a timer instruction, consider the `TON` timer instruction in the first rung of Figure 5. It has two bytes of timer-address *offset* (i.e. `0x4FCE`), which is then subtracted from the *base address* (i.e. `0x4FCE`) in the timer tuple in the configuration file (Figure 6). Hence, the `TON` instruction uses the first timer at 0^{th} index in the timer file.

`Laddis` parses the data files associated with these instructions to obtain the relevant data for decompilation.

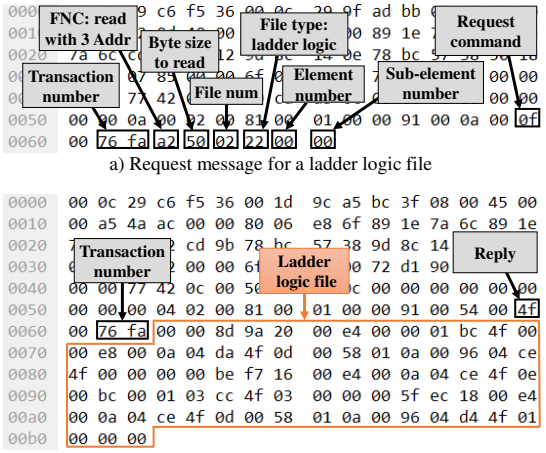
5 IMPLEMENTATION

`Laddis` is developed in Python [9] and can perform two basic functions: a) extract a binary ladder logic from a network traffic, and b) decompile the binary logic into a human-readable source code.

Extractor. The extractor component was developed using PyShark library [8], and is based on our prior experience in this area [26]. Allen Bradley Micrologix 1400-B and RSLogix 500 employ the PCCC communication protocol [1]. PCCC messages are transmitted using the EtherNet/IP protocol that adapts the *common industrial protocol* (CIP) to Ethernet. In our experiments, the PCCC message is embedded in CIP with type ID `0x91`, or `0xB1` (connected data item). When it is embedded with type ID `0x91`, the PCCC message comes at the start of the common packet format data section. When type ID `0xB1` is used, the PCCC message comes at byte offset 7 of the CIP data section after Requester ID length (1 byte), Vendor ID (2 bytes), and CIP Serial Number (4 bytes).

A ladder logic program developed by RSLogix is transferred in multiple files, including a configuration file, ladder logic file, and data files (output, input, status, timer, etc.). When a program is downloaded into a PLC, the extractor locates the files by parsing the message type (`0x0F`: request) field and function code (`0xAA`: write with 3 address fields) field (Figure 9). The start offset of file contents is flexible because the element field and the sub-element field can have 1 byte (number for 0-254) or 3 bytes (number for >=255) size. If the first byte is `0xFF`, then next two bytes represent element/sub-element number. When a ladder logic program is uploaded from a PLC, the extractor finds PCCC reply messages that contains files for a ladder logic program by monitoring request messages with function code `0xA2` (read with 3 address fields), file number, and file type. Then a reply message that has the same transaction number contains a file corresponding to the requested file (Figure 10).

Decompiler. The decompiler builds a complete human readable representation of the ladder logic program from the binary ladder



a) Request message for a ladder logic file

b) Reply message containing a ladder logic file

Figure 10: File extraction from uploading traffic

logic file outputs of the extractor program. The decompiler uses a configuration file which stores information like opcode, size, start and end addresses of various types of instructions. Based on that information it can identify the rungs, instructions and branchings. During the course of its development, we tested it with a variety of ladder logic programs to configure their instruction types. We used techniques of differential analysis to configure different types of the instructions [18]. With every subsequent test case, we changed one specific instruction of the program and analyzed the binary values. The tests were repeated until we obtained complete configuration mapping of the 120 types of instructions used in the ladder logic programs by RSLogix. We stress-tested our decompiler with many variations of RSLogix programs to find any configuration our program would have missed. We used GNU Diffutils [2] for this differential analysis. After creating a complete decompiler based on the information gained from various test datasets, we repeated more tests to find new types of instructions. The final part of the decompilation included drawing the images of the ladder logic program using the Python Image Library (PIL). The result of this program is easy to understand ladder logic program constructed solely from the network capture packets.

6 EVALUATION

6.1 Experimental Settings

Lab Setup. We use six versions of Allen-Bradley's RSLogix 500 programing software and Micrologix 1400 Series B controller. The RSLogix software (running on Windows 7 virtual machine (VM)) and Micrologix PLC communicate with each other over Ethernet. We also install Wireshark on the VM for capturing network packets.

Dataset. We developed 91 ladder logic programs of varying complexity for initial experiments and testing. However, for the evaluation, we use a different potentially-unbiased dataset consisting of 58 ladder logic source-code programs downloaded from various sources on the Internet (including github.com, plctalk.net, and theautomationstore.com). The programs are written for different physical processes (such as traffic-light, elevator and train-crossing)

Table 2: Dataset summary of ladder logic programs

File Info.		Rungs				Instructions			
File Size (kB)	# of Files	Min	Max	Total	Avg	Min	Max	Total	Avg
20-40	29	2	22	166	5.72	3	67	428	14.76
41-60	17	3	40	158	9.29	5	104	569	33.47
61-80	7	4	17	66	9.43	9	63	235	33.57
81-100	3	7	15	30	10.00	15	31	65	21.67
101-120	2	10	63	73	36.50	24	249	273	136.50
Total	58	-	-	493	-	-	-	1,628	-

involving various detectors, sensors, device controllers and counters. They have 493 rungs and 1628 instructions in total. These programs are particularly useful for performing unbiased evaluation since they are developed by different authors. Table 2 summarizes the dataset based on the file size, number of instructions and rungs.

Methodology for Experiments. A typical experiment involves RSLogix to transfer a ladder logic program to/from the Micrologix PLC, while capturing the network traffic. `Laddis` then, takes the traffic as input, extracts binary ladder logic and its associated configuration and data to separate files. It further takes the files as input and processes them to generate a decompiled ladder logic, which is then, compared with the original ladder logic program manually.

6.2 Decompilation Accuracy of Laddis

Recall that `Laddis` performs three main functions to decompile a binary ladder logic. It first 1) identifies the rungs and 2) the instructions in the rungs and then, 3) reconstructs the ladder logic from the binary logic. This section evaluates the accuracy of `Laddis` on performing these functions.

1) Identifying the Rungs. To evaluate the `Laddis` accuracy on the rungs, we analyze each ladder logic file in the dataset and count the total number of rungs and the number of instructions in each rung manually. At this stage, we only count the instructions without considering what are these instructions.

We further obtain the `Laddis` ASCII output of decompiling the files in the dataset and use a python script that processes the output of all the file and obtains the counts. Our results show that both RSLogix and `Laddis`reported exact number of rungs and instructions i.e. 493 and 1,628 in total, respectively. Hence, we conclude that `Laddis` is 100% accurate in identifying rungs.

2) Identifying the Instructions. To evaluate the `Laddis` accuracy on identifying exact instructions, we obtain the frequency of each instruction in the dataset files using RSLogix manually and a python script for `Laddis`. We found 33 unique instructions. Table 3 shows the frequency of the instructions for both RSLogix and `Laddis`. The comparison shows 100% accuracy concluding that the `Laddis` can identify instructions accurately.

3) Reconstructing the Ladder Logic. To evaluate whether `Laddis` ladder logic output conforms with the original ladder logic, we obtain `Laddis` graphical output of the ladder logic files in the dataset and compare them with the original ladder logic in RSLogix manually, using the following parameters: position of an instruction in a rung, address and configuration data in each instruction, and

Table 3: Comparison with RSLogix on decompiling

Symbols	Name	Types	RS-Logix	Laddis	Accuracy
XIC	Examine if open	Bit	521	521	100%
XIO	Examine if closed		179	179	100%
OTE	Output Energize		183	183	100%
OTL	Output Latch		20	20	100%
OTU	Output Unlatch		41	41	100%
ONS	One Shot		44	44	100%
TON	Time On Delay	Timer and Counter	58	58	100%
TOF	Timer Off Delay		5	5	100%
RTO	Retentive Timer		3	3	100%
CTU	Count Up		7	7	100%
RES	Reset		11	11	100%
EQU	Equal	Comparison	43	43	100%
GEQ	Great Than or Equal		20	20	100%
GRT	Greater Than		6	6	100%
LEQ	Less Than or Equal		5	5	100%
LES	Less Than		8	8	100%
LIM	Limit Test		14	14	100%
MEQ	Masked Comparison for Equal		2	2	100%
NEQ	Not Equal		7	7	100%
ADD	Add	Math	13	13	100%
SUB	Subtract		14	14	100%
MUL	Multiply		1	1	100%
DIV	Divide		4	4	100%
CLR	Clear		4	4	100%
SCP	Scale with Parameters		1	1	100%
MOV	Move	Data Handling	92	92	100%
MVM	Masked Move		2	2	100%
OR	Logical OR	Branch	202	202	100%
JSR	Jumpt to Subroutine	Program Control	43	43	100%
BSL	Bit Shift Left	Application Specific	5	5	100%
SQO	Sequencer Output		4	4	100%
MSG	Message	SLC Communication	8	8	100%
END	End	Unspecified	58	58	100%
	Total		1628	1628	100%

the connection of two instructions whether in series (AND logic) or parallel (OR logic). Figures 11a and 11b shows an example of RSLogix and Laddis that are used for comparison. Our comparison results show that RSLogix source code and Laddis graphical output comply with each other, concluding that Laddis can reconstruct the source code of a ladder logic from its binary representation accurately.

6.3 Laddis Compatibility with Older RSLogix

We use RSLogix 9.00.04 version to perform the experiments on Laddis accuracy in the last section. We further repeat the experiments on five older versions of RSLogix to evaluate the compatibility of Laddis with these version. Table 4 shows the overall results.

Table 4: Laddis compatibility with older RSLogix versions

RSLogix Versions	Number of Files	Laddis Accuracy(%)
8.10.00	28	100%
8.20.00	35	100%
8.30.00	40	100%
8.40.00	58	100%
9.00.00	58	100%
9.00.04	58	100%

We did not find any discrepancies in the Laddis output. Thus, we conclude that Laddis is compatible with the older versions.

6.4 Detailed Attack Scenarios

This section evaluates the effectiveness of Laddis under the three attack scenarios outlined earlier.

Assumption. In the first two scenarios, we assume that the attacker can intercept and modify the network traffic between an engineering workstation and a target PLC using man-in the-middle (MITM) [6]. In ICS, MITM is a known threat. Kaspersky Lab reported that 91.6% of the ICS environments (that they analyzed) use insecure protocols that are prone to data interception and modification using MITM [5]. In the past, MITM was observed in the ICS operational environments such as Stuxnet [15] and IRONGATE [4]. The third attack scenario does not involve MITM.

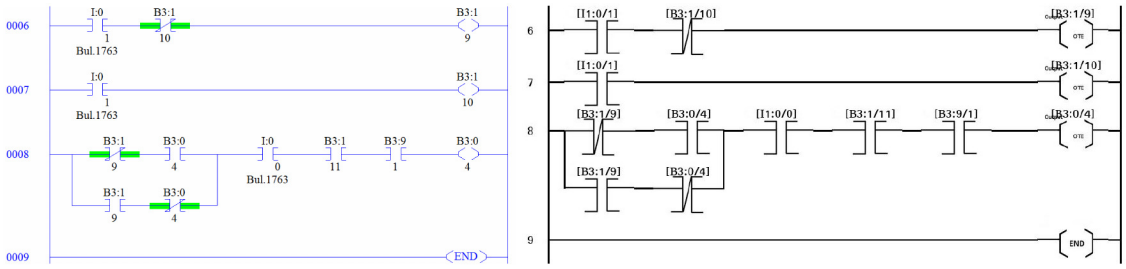
1) DEO Attack 1: Hiding infected ladder logic (running in a target PLC) from RSLogix.

Attack Scenario: The attacker transfers an infected ladder logic to a target PLC. To hide the infection, when a control operator/engineer tries to acquire the program from the PLC using RSLogix, the attacker intercepts the traffic via MITM and replaces the infection with the original logic. Consequently, the RSLogix shows the original (uninfected) ladder logic to the operator (Figure 1).

Attack Execution: We use traffic-light ladder logic program, which consists of three timers, each controlling one of the three signal lights (green, orange, and red). The goal of the attacker is to make consistent change in the timing of green light so that the light remains on for 100 seconds, instead of original 5 seconds.

We employ infamous ARP poisoning using Ettercap [23] to achieve MITM. When a control engineer downloads the traffic-light program to a target PLC using RSLogix, the attacker intercepts the network traffic, and the *preset* value of the timer controlling the green light from 5 seconds (original value) to 100 seconds (attacker's desired value) (Figure 12). Now the PLC turns on the green light for 100 seconds. Recall that a timer instruction has three parameters: base, preset, and accumulated. The *preset* sets the timing. At this stage, if RSLogix uploads the program from the PLC, it will show the attacker's change in the timer instruction. Thus, the attacker also intercepts the upload traffic and replaces the 100 seconds with 5 seconds in the timer instruction, which hides the infection from RSLogix (shown in Figure 13).

Forensic Analysis with Laddis: Strange ARP table (due to ARP poisoning) or duplicated packets (due to the MITM) may raise suspicion that can lead to a forensic investigation. If the network is captured, a forensic investigator can filter the traffic based on source



(a) RSLogix source code (b) Laddis Graphical output

Figure 11: Comparison of RSLogix source code and Laddis Graphical output

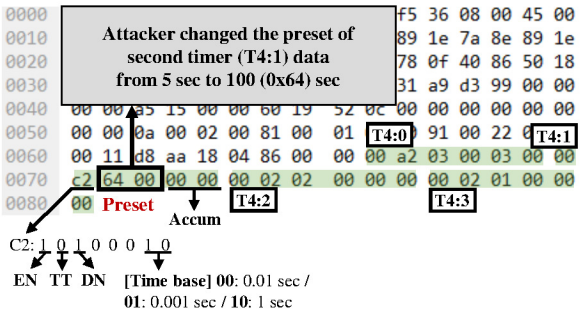


Figure 12: Modifying timer data in downloading traffic

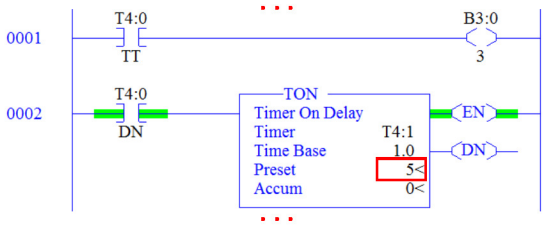


Figure 13: RSLogix cannot show the program actually running in PLC

and destination MAC addresses and then, use Laddis to extract two versions of the ladder logic program. First is the original that is exchanged between RSLogix and the attacker and the second is exchanged between the attacker and the PLC. Laddis further decompiles both ladder logic programs. Figure 14 shows Laddis ASCII output and points to the differences in the timer instruction.

2) DEO Attack 2: Crashing the Programing Software.

Attack Scenario: The attacker infects the ladder logic in a target PLC. The goal of this attack is to create a denial of service when the programing software tries to acquire the logic from the PLC. In particular, the attacks achieves to crash the software while uploading the ladder logic from the PLC (Figure 2).

Attack Execution: We use ARP poisoning (with Ettercap) for MITM and the traffic light program to execute this attack. To crash RSLogix, the attacker replaces an instruction with unexpected byte codes in the ladder logic program while it is being transferred to RSLogix. Figure 15(a) depicts the original piece of ladder logic code

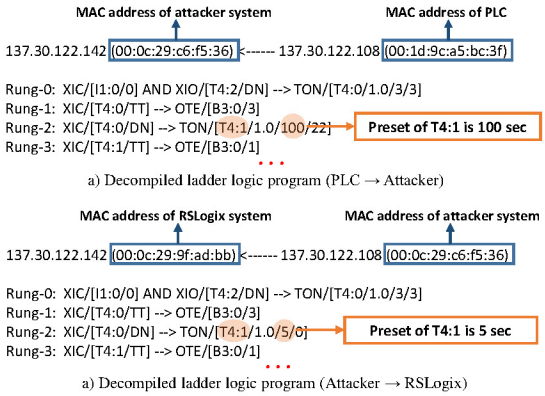


Figure 14: Laddis ASCII output: attack scenario #1

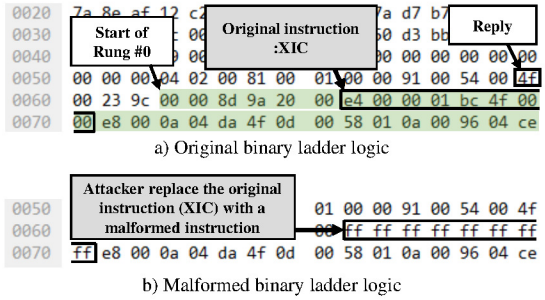


Figure 15: Modifying ladder logic program

in a network packet transferring from the PLC to RSLogix. The attacker intercepts the packet and replaces the XIC instruction in the first rung with 8 bytes of '0xFF' byte code (Figure 15(b)) and then, forwards the malformed packet to the RSLogix, which apparently crashes the software.

Forensic Analysis with Laddis: A forensic investigator can use Laddis to extract the *original* (from PLC to the attacker) and *malformed* (from the attacker to RSLogix) ladder logic programs from the network traffic. Laddis can further decompiles the programs. The malformed logic is challenging. However, Laddis is equipped to identify the ladder logic instruction while ignoring the malformed or corrupted logic. Figure 16 shows the Laddis output. Laddis ignores the first rung that has the corrupted (0xFF

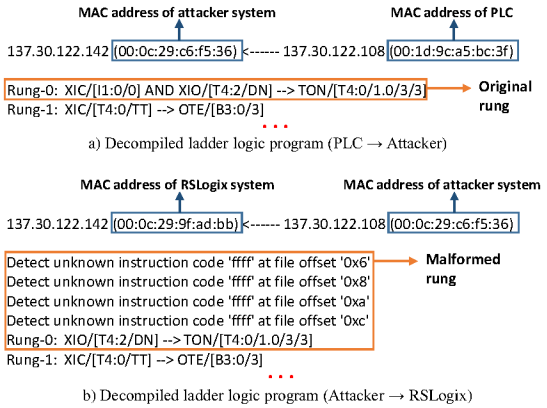


Figure 16: `Laddis` ASCII output: attack scenario #2

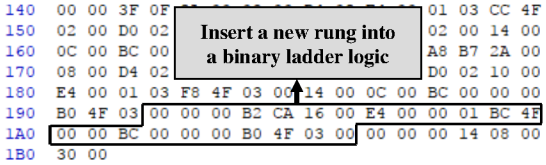


Figure 17: Inserting a new rung into the ladder logic

sequence of bytes in the) instruction and decompiles the second rung successfully.

3) DEO Attack 3: Injecting a crafted ladder logic program to a PLC that crashes the programing software.

Attack Scenario: The goal of this attack is to crash the programing software when it attempts to acquire the ladder logic from an infected PLC *without* needing to maintain a MITM posture. In this case, the attacker creates a specially crafted ladder logic program that runs successfully on a PLC, but when the programing software attempts to acquire the logic, the program crashes the software (Figure 3). This is the stealthiest attack among the three attack scenarios and only requires to update the PLC once with the manipulated program. The attacker can achieves it by infiltrating into the ICS network or obtaining local access to the PLC.

Attack Execution: To perform this attack, we first craft a ladder logic program. In particular, we insert a new (malicious) rung before the last rung (which contains only END instruction) of the traffic-light program. It is done at the binary-level, since RSLogix or any other ladder logic programming software does not allow to make certain changes due to incompatibility. Figure 17) shows a sequence of bytes that is the new rung at binary-level.

Adding a rung influences both size and contents of the ladder logic file, and we found six byte locations related to ladder logic size or ladder logic integrity in the configuration file (type:0x03) and in an unknown file (type:0x24). Figure 18(a) depicts a configuration file which has type '0x03'. In the configuration file, byte location 0x88 - 0x89, two bytes right after the file type 0x22 (ladder logic file), represents size of the ladder logic file. And we also found that the value of location 0x7a - 0x7b is unknown, but it varies as size of ladder logic file. Meanwhile, the values of location 0x36 - 0x37

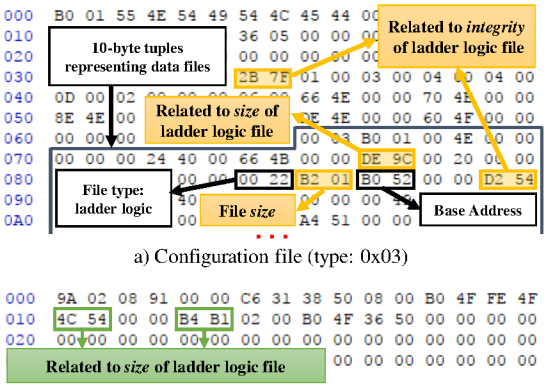


Figure 18: Byte locations related to the integrity of ladder logic

and location 0x8e - 0x8f are related to integrity of ladder logic file. They are changed according to contents of ladder logic files even if size are the same. If we do not correct these values according to changed ladder logic files, the PLC always refused to accept the program. On the other hand, Figure 18(b) depicts an unknown file which has type '0x24'. We also found two byte locations, 0x10 - 0x11 and 0x14 - 0x15, changed according to size of ladder logic files.

If we correct all these values, the modified ladder logic programs runs in PLCs and also can be acquired by RSLogix without crashing. However, if we correct the values only in the configuration file and do not change the values in the other file (type 0x24), this semi-valid programs runs successfully in PLCs but RSLogix fails with an "Unknown error" message.

Forensic Analysis with Laddis: To investigate this attack, the investigator can use RSLogix to attempt to acquire the crafted ladder logic program from the PLC. It crashes the RSLogix, but apparently, makes the PLC to send the entire ladder logic to RSLogix over the network. If the network traffic is captured, the `Laddis` will extract the ladder logic program from the network traffic dump and can further decompile it to show the infected code that has an additional rung in the end of the logic. `Laddis` is able to decompile the crafted program because it concentrates on the actual contents of the program without depending on non-essential metadata for decompiling (Figure 19).

6.5 Performance Measurement

We carried out performance tests for each extracting part and decompiling part of `Laddis` with 58 pcap files. We ran experiments on a Virtual machine running Ubuntu 16.04 on a single Intel 3.40 Ghz core with 4GB of RAM. Execution time for extracting files of ladder logic programs was directly proportional to pcap file size and the number of packets in a file (Table 5). On the other hand, execution time of decompiling part of `Laddis` is affected by the number of rungs in extracted ladder logic programs (Table 6 in Appendix) and the number of instruction in the programs (Table 7 in Appendix).

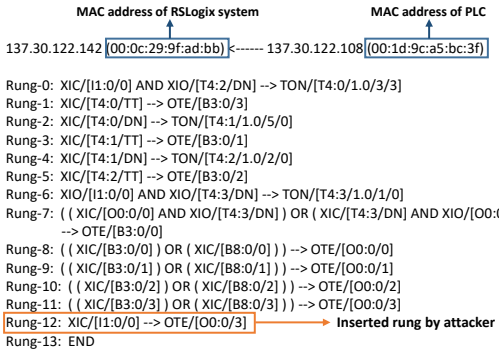


Figure 19: `Laddis` **ASCII output: attack scenario #3**

Table 5: Extractor performance vs. *pcap* size

FileSize (KB)	# of Files	Avg.#of Packets	Max # of Packets	Avg Time (sec)	Max Time (sec)
0-100	14	3,663	10,288	4.517	11.51
101-300	11	4,602	17,414	5.86	22.74
301-400	8	3,033	6,164	3.78	7.69
401-500	8	4,017	5,143	5.03	6.50
501-600	10	1,625	4,959	2.07	6.05
>=601	7	2,305	7,721	3.15	10.20

7 RELATED WORK

PLCs are built with safety and endurance as primary requirements, and very few private protocols have built-in security features [24]. Patel et al acknowledge that SCADA devices using secret proprietary protocols can be easily targeted by attackers attempting to reverse engineer those protocols. Our study is an example of reverse engineering of PCCC protocol used by many popular Allen Bradley PLCs [1]. Narayan et al mention that unspecified protocols exist in different OSI models [22]. These protocols are not immune to reverse engineering techniques and attackers can extract communication data from a SCADA network by decoding such protocols. Authors mention that task of protocol reverse engineering is "tedious" and "time consuming" manually. There are disassembly tools like IDA Pro [3] to perform the disassembly of binary executable; however, analysts [28] mention that its disassembly is prone to errors and requires further manual analysis. Our tool provides a simpler way to analyze the malicious code without the need for access to the PLC.

Fovino et al. [16] propose the SCADA protocol sensor, which is analogous to Snort [25]. The authors use the network analysis of the traffic to detect malicious code in the devices using DNP3 and Modbus protocols. They use databases to store the signature of the single packet-based attacks, maintain the status of network devices, use state validation based on another database which includes the rules that system adopts during the critical control.

Cheung et al. proposed model-based intrusion detection system for SCADA network. They construct normal model that characterize the expected behavior of SCADA network, and detect attacks that deviate from the constructed normal model [13]. Derarnan et al. emphasize the importance of network behavioral analysis and warn that signature-based detection systems can fall short in case of zero-day attacks [14]. These authors have based their research on securing the SCADA networks from intrusions through the implementation of protocol reversing, misuse detection, anomaly detection and the successful analysis of network traffic. Our tool constructs the actual ladder logic program from the network traffic and in case of attack without reliance on the HMI to detect the changes; it also provides the extracted malicious code.

The discovery of Stuxnet attracted huge worldwide interest in the field of industrial control system security [21]. When the Iranian nuclear facilities using Siemens SIMATIC WinCC SCADA systems were compromised, it was evident that the further study in securing the critical infrastructures is crucial [12].

As discussed in Kotler et al. [20], running additional software to check the programs in a PLC has performance drawbacks. They used finite state machine to check computational tree logic and linear temporal logic to detect malicious or faulty PLC programs. They show how small changes induced by intrusions can go undetected in the ladder logic software. Their method analyzed minute malfunctions in ladder logic programs running a virtual crane, and found unnoticeable changes that could cause catastrophic outcomes [20]. Similarly, Valentine & Farkas state that an attacker can remove an instruction like a coil needed to trigger an alarm but the code will compile in the Human Machine Interface (HMI) and will be downloaded to the PLC [27]. The authors propose code validation process using a database to flag malicious code.

Cutter [26] is a parsing tool for data files exchanged between a PLC and engineering workstation. It is developed by the authors. Cutter is not a decompiler and has mainly developed for the data files.

8 CONCLUSION

In this work, we introduced a new class of attacks on industrial control systems–*denial of engineering operations*, which can severely impact the situational awareness of the operator, and can provide time and cover for a malicious control logic installed on a PLC to produce the effects in the physical world.

Fundamentally, the vulnerability stems from the fact that engineering software used to develop, test, and deploy the control logic expects any program retrieved from the PLC to be properly formatted and there are few (if any) provisions to recover from a malformed responses that fail integrity checks. At the same time, the PLC has fewer such verifications, which allows it to execute the malformed code. This dichotomy allows an adversary who has access to the PLC via the network the opportunity to deceive the operator by crafting and installing programs that function as desired but not understood by the engineering software. Although, it is possible to overcome this by reprogramming the PLC with a known-good version of the code; this would also destroy all the traces of the attack for analytical purposes.

To address the problem, we developed `Laddis`–a ladder logic decompiler that can correctly reconstruct the source of the original code from a network trace. This allows the operator to quickly analyze the attack and respond accordingly. Future extension can automate this process and allow defensive reaction in real time.

APPENDIX

Table 6: Decompiler performance vs. number of rungs

# of Rung	# of Files	Max # of Rung	Avg time (sec)	Max time (sec)
0-3	8	3	0.16	0.19
4-5	18	5	0.30	0.58
6-9	11	9	1.26	3.15
10-14	16	13	1.01	2.60
>=15	5	63	5.46	16.41

Table 7: Decompiler performance vs. number of inst.

# of Inst.	# of Files	Max # of Inst.	Avg time (sec)	Max time (sec)
0-10	8	9	0.176	0.36
11-20	16	20	0.356	1.19
21-30	9	27	0.922	1.55
31-40	14	37	0.897	2.60
41-60	5	60	1.509	3.15
>=61	6	312	4.738	16.41

Table 8: Instruction Information

Op-code	Byte code	Size (byte)	Op-code	Byte code	Size (byte)	Op-code	Byte code	Size (byte)
END	0030	2	MUL	00A4	22	LBL	00EC	8
XIC	00E4	8	DIV	00A8	22	JSR	0054	10
XIO	00E8	8	SQR	0118	16	RET	0024	2
OTE	00BC	8	NEG	0078	16	SBR	00F4	10
OTL	00C0	8	TOD	0148	16	TND	002C	2
OTU	00C4	8	FRD	014C	16	MCR	0020	2
ONS	02AC	8	GCD	02BC	16	SUS	007C	10
OSR	0278	16	MOV	0070	16	ABL	01E0	16
OSF	0274	16	MVM	0098	22	ACB	01E4	16
TON	0158	10	AND	008C	22	ARD	0200	22
TON	003C	10	OR	0090	22	ARL	0204	22
TON	029C	10	XOR	0094	22	AWT	0210	22
TOF	0154	10	NOT	006C	16	AWA	020C	22
TOF	0040	10	CLR	0050	10	AHL	01F8	28
TOF	0298	10	COP	0088	22	ACL	01F0	22
RTO	0150	10	FLL	0084	22	ACN	01EC	22
RTO	0038	10	DLG	02B0	10	ACI	01E8	16
RTO	028C	10	SCL	0114	28	AIC	01FC	16
CTU	0044	10	INT	012C	10	AEX	01F4	28
CTD	0048	10	STS	0290	10	ASC	0208	28
RES	004C	10	PID	027C	22	ASR	0214	16
RHC	035C	16	PTO	0280	10	HSL	026C	34
RTA	02B4	2	PWM	0284	10	RAC	0288	16
IIM	0174	22	UID	02A0	10	SIN	024C	20
IOM	0178	22	UIE	02A4	10	COS	0248	20
SVC	0294	10	UIF	02A8	10	TAN	0250	20
MSG	0270	16	CPW	02B8	22	ASN	0240	20
MSG	02CC	22	RCP	02C0	16	ACS	023C	20
REF	0120	2	LCD	02C8	46	ATN	0244	20
LIM	00FC	22	RPC	037C	16	LN	0234	24
MEQ	00E0	22	BSL	00B0	22	LOG	0238	24
EQU	00C8	16	BSR	00AC	22	DEG	0264	24
NEQ	00CC	16	SQC	00B8	28	RAD	025C	16
LES	00D8	16	SQL	0100	22	XPY	0230	22
GRT	00D0	16	SQO	00B4	28	ABS	0260	16
LEQ	00DC	16	FFL	0104	22	SCP	0254	40
GEQ	00D4	16	FFU	0108	22	SWP	0258	16
CPT	0228	20	LFL	010C	22	DCD	0080	16
ADD	009C	22	LFU	0110	22	ENC	0190	16
SUB	00A0	22	JMP	0058	10	TDF	0360	22

REFERENCES

[1] 1996. DF1 Protocol and Command Set Reference Manual. http://ow.ly/N61S30fsdqg. (1996). [Online; accessed 23-Sept-2017].

[2] 2017. GNU Diffutils. https://www.gnu.org/software/diffutils/. (2017). [Online; accessed 23-Sept-2017].

[3] 2017. Hex-Rays. https://www.hex-rays.com/. (2017). [Online; accessed 23-Sept-2017].

[4] 2017. IRONGATE ICS Malware. https://www.fireeye.com/blog/threat-research/2016/06/irongate_ics_malware.html. (2017). [Online; accessed 23-Sept-2017].

[5] 2017. Kaspersky. https://www.kaspersky.com/blog/industrial-vulnerabilities/12596/. (2017). [Online; accessed 23-Sept-2017].

[6] 2017. Man-in the middle attack in ICS. https://ics-cert.us-cert.gov/content/overview-cyber-vulnerabilities#man. (2017). [Online; accessed 23-Sept-2017].

[7] 2017. PLCS.net. http://www.plcs.net/downloads/index.php?&direction=0&order=&directory=Allen_Bradley. (2017). [Online; accessed 23-Sept-2017].

[8] 2017. Python Package Index Pyshark. https://pypi.python.org/pypi/pyshark. (2017). [Online; accessed 23-Sept-2017].

[9] 2017. Python Software Foundation. https://www.python.org/. (2017). [Online; accessed 23-Sept-2017].

[10] I. Ahmed, S. Obermeier, S. Sudhakaran, and V. Roussev. 2017. Programmable Logic Controller Forensics. *IEEE Security Privacy* 15, 6 (November 2017), 18–24. https://doi.org/10.1109/MSP.2017.4251102

[11] Irfan Ahmed, Vassil Roussev, William Johnson, Saranyan Senthivel, and Sneha Sudhakaran. 2016. A SCADA System Testbed for Cybersecurity and Forensic Research and Pedagogy. In *Proceedings of the 2Nd Annual Industrial Control System Security Workshop (ICSS '16)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/3018981.3018984

[12] T. M. Chen and S. Abu-Nimeh. 2011. Lessons from Stuxnet. *Computer* 44, 4 (April 2011), 91–93.

[13] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. 2007. Using Model-based Intrusion Detection for SCADA Networks. (jan 2007), 127–134.

[14] M. Deraman, J. M. Desa, and Z. A. Othman. 2010. Multilayer packet tagging for network behaviour analysis. In *2010 International Symposium on Information Technology*, Vol. 2. 909–913.

[15] Nicolas Falliere, Liam O Murchu, , and Eric Chien. 2011). *W32.Stuxnet Dossier*. Technical Report. Symantec.

[16] I. N. Fovino, A. Carcano, T. D. L. Murel, A. Trombetta, and M. Masera. 2010. Modbus/DNP3 State-Based Intrusion Detection System. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 729–736.

[17] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *24th Annual Network & Distributed System Security Symposium (NDSS)*.

[18] Simson Garfinkel, Alex Nelson, and Joel Young. 2012. A General Strategy for Differential Forensic Analysis. In *The Digital Forensic Research Conference DFRWS*. S50–S59.

[19] Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. 2017. On Ladder Logic Bombs in Industrial Control Systems. *CoRR* abs/1702.05241 (2017). http://arxiv.org/abs/1702.05241

[20] S. Kottler, M. Khayamy, S. R. Hasan, and O. Elkeelany. 2017. Formal verification of ladder logic programs using NuSMV. In *SoutheastCon 2017*. 1–5.

[21] R. Langner. 2011. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security Privacy* 9, 3 (May 2011), 49–51.

[22] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. 2015. A Survey of Automatic Protocol Reverse Engineering Tools. *ACM Comput. Surv.* 48, 3, Article 40 (dec 2015), 26 pages.

[23] A. Ornaghi and M. Valleri. 2017. Ettercap. https://ettercap.github.io/ettercap/. (2017). [Online; accessed 23-Sept-2017].

[24] Sandip C. Patel, Ganesh D. Bhatt, and James H. Graham. 2009. Improving the Cyber Security of SCADA Communication Networks. *Commun. ACM* 52, 7 (July 2009), 139–142.

[25] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[26] Saranyan Senthivel, Irfan Ahmed, and Vassil Roussev. 2017. SCADA network forensics of the PCCC protocol. *Digital Investigation* 22 (2017), S57–S65.

[27] S. Valentine and C. Farkas. 2011. Software security: Application-level vulnerabilities in SCADA systems. In *2011 IEEE International Conference on Information Reuse Integration*. 498–499.

[28] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*. 158–177.