University of New Orleans

ScholarWorks@UNO

University of New Orleans Theses and Dissertations

Dissertations and Theses

Summer 8-6-2018

Semantic-aware Stealthy Control Logic Infection Attack

Sushma kalle

University of New Orleans, New Orleans, sushmakalle2@gmail.com

Follow this and additional works at: https://scholarworks.uno.edu/td

Part of the Information Security Commons

Recommended Citation

kalle, Sushma, "Semantic-aware Stealthy Control Logic Infection Attack" (2018). *University of New Orleans Theses and Dissertations*. 2512.
https://scholarworks.uno.edu/td/2512

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

Semantic-aware Stealthy Control Logic Infection Attack

A Thesis

Submitted to Graduate Faculty of the
University Of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Information Assurance

by

Sushma Kalle

B.Tech., N.B.K.R.I.S.T., 2012

August 2018

Acknowledgment

I would like to thank my advisor, Dr. Irfan Ahmed, for the constant support and being the best guide through out my study.

I would like to thank Dr. Vassil Roussev and Dr. Minhaz Zibran for serving in my thesis Defense Committee

My heartfelt thanks to Hyunguk Yoo and Nehal Ameen for all the help and support.

My special thanks to my family and amazing friends who could make me feel like anything is possible and made me the person I am today.

This thesis work is wholeheartedly dedicated to my mother and sister who made my life worth living.

To Mr. Raghava Krishna Akshintala, for all the advice and counsel, which made me stand at where I am today.

To my uncle and aunt, Mr. Kalyan Kalle and Mrs. Mallika Kalle, who continually provide their moral, spiritual, emotional and financial support.

Contents

List of Figures v

List of Tables vi

List of Algorithms vii

Abstract viii

1 Introduction 1

2 Related Work 4

3 Control Logic Infection Attack (CLIK) 8
- 3.1 Phase I: Stealing the Original Control Logic from a Target PLC . . . 8
- 3.2 Phase II: Decompiling the Stolen Binary to Source Code . . . 9
- 3.3 Phase III: Infecting the Control Logic via Rule-based Approach . . . 10
- 3.4 Phase IV: Concealing the Malicious PLC Control Logic from Engineering Software . 13

4 Real-world Implementation of CLIK 18
- 4.1 Subverting PLC Password Protection by Exploiting a Zero-day Vulnerability . . . 18
- 4.2 Control Logic Uploader & Downloader . . . 22
- 4.3 IL Decompiler & Compiler . . . 23
- 4.4 Mapping RX630 Opcodes with IL Instructions . . . 27
- 4.5 Malicious Logic Generator . . . 27
- 4.6 Virtual M221 PLC . . . 32

5 Evaluation 35
- 5.1 Experimental Settings . . . 35
- 5.2 Reliability of the Password Attack . . . 35
- 5.3 Compilation & Decompilation Accuracy . . . 36
- 5.4 Generation of Malicious Logic . . . 36
- 5.5 Effectiveness of the M221 Virtual PLC . . . 38
- 5.6 Putting CLIK-Phases All Together . . . 39

6 Conclusion 40

Bibilography 41

Vita 43

List of Figures

2.1 Infection hiding based on a packet filter 6

3.1 High-level overview of CLIK, an automated control logic infection attack 9
3.2 Decompilation of the Control Logic 10
3.3 A systematic approach to build a virtual PLC 15

4.1 Normal password authentication between the Schneider Electric Modicon M221 PLC and SoMachine-Basic engineering software 19
4.2 Password authentication while exploit the vulnerability in the Schneider Electric Modicon M221 PLC by the `CLIK` 19
4.3 Address space of the M221 PLC 20
4.4 Start address and size of M221 control logic blocks 23
4.5 Simple Operation Block 25
4.6 Complex Operation Block 25
4.7 Mapping RX630 Opcodes with IL Instructions 27
4.8 Mapping opcode with Instructions 29
4.9 Similar Opcode Examples 30
4.10 A snippet of aligned M221 protocol message chunks 32
4.11 Read and write message format of the M221 proprietary protocol 33
4.12 Session ID request/response message format of the M221 proprietary protocol 33
4.13 Integrity check of control-logic using a checksum in M221 proprietary protocol 34

List of Tables

4.1 Instruction List representation and corresponding machine code description 28

5.1 The experimental results of the password reset attack 36
5.2 The accuracy of the decompiler `Eupheus` 37
5.3 The experiment results on the virtual M221 PLC 38
5.4 The final evaluation result on `CLIK` 39

List of Algorithms

1 Pseudocode for password reset attack 22

Abstract

In this thesis work we present CLIK, a new, automated, remote attack on the control logic of a programmable logic controller (PLC) in industrial control systems. The CLIK attack modifies the control logic running in a remote target PLC automatically to disrupt a physical process. We implement the CLIK attack on a real PLC. The attack is initiated by subverting the security measures that protect the control logic in a PLC. We found a critical (zero-day) vulnerability, which allows the attacker to overwrite password hash in the PLC during the authentication process. Next, CLIK retrieves and decompiles the original logic and injects a malicious logic into it and then, transfers the infected logic back to the PLC. To hide the infection, we propose a virtual PLC that engages the software the virtual PLC intercepts the request and then, responds with the original (uninfected) control logic to the software.

KEY WORDS

Programmable Logic Controller, Instruction List, Decompiler, Industrial Control System, Digital Forensics, Cyberattack

Chapter 1

Introduction

Industrial control systems (ICS) are used to automate physical processes such as wastewater treatment plant, gas pipeline, and power grid station. These systems are increasingly connected to corporate network and Internet for significant economic gain. Unfortunately, the connectivity also makes them vulnerable to cyberattacks [6]. To secure ICS environments, it is imperative to understand their threat vector, i.e., how an adversary can target these systems.

An ICS environment consists of a control center and field sites [7,8]. The physical processes are located at field sites and are monitored and controlled via sensors, actuators, and programmable logic controllers (PLCs). The PLCs send data to ICS services at the control center such as human-machine-interface (HMI), Historian and Engineering Workstation. They are programmed to define a control logic to maintain the desired state of a physical process. For instance, in a gas pipeline, a PLC monitors and controls the gas pressure of the compressed gas in the pipe. The control logic of the PLC is defined as follows: when the gas pressure exceeds a certain threshold, the PLC opens a solenoid valve (i.e., an actuator) to release some gas, which reduces the gas pressure in the pipe.

An attacker targets the control logic to compromise a PLC to sabotage a physical process. For instance, Stuxnet [9] infects the control logic of the Siemens S7-300 PLCs controlling variable frequency drives of centrifuges. The infected logic disrupts the normal operation of the drives by changing their motor speed periodically from 1,410 Hz to 2 Hz to 1,064 Hz and then over again.

A typical attack of infecting the control logic involves gaining access to the engineering software at the control center, modifying the logic and then, transfer it to a target PLC. In particular, Stuxnet [9] employs this approach. It compromises `STEP 7` engineering software (by replacing `s7otbxdx.dll` that handles communication with the PLC) and infect the PLC with a malicious control logic.

In this thesis work, we present `CLIK`, an autonomous attack on the control logic of a PLC. We assume a realistic scenario where an engineering software in the control center is not accessible for

the attack, thereby making the attack more challenging. In particular, the attack cannot utilize the engineering software to do the following: 1) transfer the control logic to/from the PLC, 2) use the project files of the current control logic of a target PLC residing at the engineering workstation to make malicious modification, and 3) provide the security credentials (such as password) to the PLC to access the control logic.

The goal of the `CLIK` attack is to introduce malicious logic in a target PLC automatically without access to the engineering software. The `CLIK` attack is initiated after the attacker penetrates into an ICS network and can send/receive messages to/from a target PLC remotely. The compromising of ICS network is out of the scope of this work and can be achieved via typical attack vector in our IT world such as infected external USB device, vulnerable web server, etc.

`CLIK` is a full attack that involves compromising the PLC security measures followed by stealing the control logic from the PLC, decompiling the stolen (compiled) binary of the control logic to inject the malicious logic, and then transferring the infected binary back to the PLC. Finally, to hide the malicious logic in the PLC from the engineering software, we propose a new approach consisting of a virtual PLC and captured network traffic of original control logic. When the engineering software attempts to acquire the control logic from the PLC, the virtual PLC intercepts the request and then, responds by sending the original control logic using the captured traffic.

To present a real-world case study, we implement the full `CLIK` attack on a real PLC (Schneider Electric Modicon M221) and engineering software (SoMachine-Basic) used in industrial settings. The PLC protects its control logic using password authentication. The attacker can employ different known techniques to compromise the password such as brute-force and social engineering attacks. However, in this work, we present a critical (zero-day) vulnerability in the password authentication mechanism of the PLC. We demonstrate that the attacker can exploit the vulnerability to reset the password by overwriting the original password hash in the PLC with its password hash during the authentication process. We have already disclosed the vulnerability responsibly to the PLC vendor. After successful password authentication, the PLC allows reading the control logic remotely.

Since SoMachine-Basic supports Instruction List (one of the languages defined by IEC 61131-3) to program the PLC, the acquired control logic from the PLC is a binary (compiled) form of an Instruction List (IL) program. We developed a decompiler `Eupheus` for the Instruction List to transform the binary into its corresponding IL source code. We further utilize a rule-based approach

to define criteria for automated malicious modification of the source code, and then compile the code to transform it into a binary that can run on the PLC. Finally, the binary is transferred back to the PLC to compromise the control logic. To hide the malicious logic in the PLC from SoMachine-Basic, we developed a virtual PLC that engages the SoMachine-Basic using the captured traffic of the original control logic. It is worth mentioning that we only use SoMachine-Basic to write and install an uninfected control logic in the PLC (which is a part of normal engineering operations), and evaluate the virtual PLC during the attack.

We evaluate the `CLIK` implementation on 52 control logic programs including three real-world control logic. The evaluation results show that `CLIK` exploits the zero-day vulnerability to reset the passwords successfully and then, retrieves the control logic remotely. The decompiler `Eupheus` decompiles the binary control logic to IL code with 100% accuracy. `CLIK` is evaluated with infection rules, and the infected logic are transferred to the PLC with 100% success rate. Finally, the virtual PLC hides the infected PLC from the SoMachine-Basic successfully.

Organization. The rest of the work is organized as follows: Chapter 2, Related work presents the related work. Chapter 3, Control logic Infection Attack (CLIK) presents the detailed explanation about the four phases of the attack. Chapter 4, Real-world Implementation of CLIK presents the implementation details CLIK in various phases and the zero day vulnerability we have discovered and its exploitation, followed by chapter 5, Evaluation with the evaluation results and ends with a conclusion in Chapter 6 Conclusion.

Chapter 2

Related Work

This chapter presents the attacks on PLCs that are closely related to `CLIK`.

Pavlovic *et al.* [13] present a tool for the fully automated transformation of IL programs into models of the NuSMV model checker. Both this work and `CLIK` use IL programs. However, `CLIK` is different in that it decompiles a low-level representation of a control logic to IL program.

A formal verification of PLC software is of great importance, since the safety demands of many systems based on PLC are considerable. Which is why our decompiler `Eupheus`is designed to verify that the control logic downloaded to the PLC is authentic and malicious logic free, by decompiling the machine code into higher level Instruction Language, so that anyone can conduct forensic investigation easily in case of any suspicion.

McLaughlin [12] evaluates how PLC malware may infer the structure of the physical plant and then, use this information to construct a dynamic malicious payload to achieve an adversary's end goal. He finds that a dynamic payload that causes unsafe behavior for an arbitrary process definition can be constructed. The malware first gathers clues from within the control system regarding either the nature of the process, the layout of the physical plant, or both. These clues are then used to generate a payload that can be uploaded to the PLC and executed. His focus was autonomously generating a PLC payload. However, `CLIK` focuses on infecting a current control logic running by a target PLC in a ICS environment. It uses rule-based approach based on heuristics instead of the exact knowledge of a target physical process.

McLaughlin *et al.* [11] also present SABOT, a tool that automatically maps the control instructions in a PLC to an adversary-provided specification of the target control systems behavior using a dynamically generated payload. This approach requires that the adversary has to correctly specify full system behavior to ensure complete and correct payload compilation. It is not for adversaries that do not understand the behavior of the victim plant [11]. This work is different from `CLIK` in that CLIK's malicious logic generator does not assume any knowledge of the physical process. The

logic generator is configured based on heuristics and the semantics of variables and IL instructions in control logic and thus, modify the analog values of the control logic. The attacker could know how the target physical system behaves, or just use some general feature and use that information to configure the malicious logic generator to find and modify the specific variable related to the target variable and change the the flow of the entire control logic, just by changing the operation block equation, or even replacing one operator by its opposite, and so, make the control flow act the exact opposite way to the way it was originally intended.

Abbasi *et al.* [5] looked into the current state of host-based detection techniques for embedded devices, with a particular focus on PLCs. They only mimicked a PLC environment and used Codesys as the PLC run-time. They implemented an attack against a PLC environment by exploiting the run-time configuration of the I/O pins used by the PLC to control a physical process. They present two variations of the attack implementation. The first implementation allows an extremely reliable manipulation of the process at the cost of requiring root access. The second implementation slightly relaxes the requirement of reliable manipulation while allowing the manipulation to be achieved without root access. `CLIK` on the other hand does not exploit the configuration of the I/O pins. Instead, it modifies the control logic of a target PLC directly. Furthermore, it implements a virtual PLC utilizing the network traffic of a normal control logic to engage an engineering software.

The attack allows one to reliably take control of the physical process normally managed by the PLC,while remaining stealth to both the PLC runtime and operators monitoring the process through a Human Machine Interface, a goal much more challenging than simply disabling the process control capabilities of the PLC, which would anyway lead to potentially catastrophic consequences. The attack does not require modification of the PLC logic or traditional kernel tampering or hooking techniques, which are normally monitored by anti-rootkit tools

We implement the `CLIK` attack on a real PLC used in industrial settings. `CLIK` can look for special instructions in the decompiled code to detect specific engineering software. We generate and apply 3 rules to manipulate and infect the PLC, using 3 PLC programs that can be used in a variety of situations. The first infection changes the output or input instructions to either energize the output or not. The second infection inserts a new rung into the Instruction list. The third infection modifies the analog flow determinant and changes the operation block.

Garcia *et al.* [10] present Harvey, a PLC rootkit that implements a physics-aware stealthy man-

in-the-middle attack against power grid control systems. They implemented the attack on Allen Bradley PLC and evaluated it on a real-world electric power grid test-bed. They modified the firmware of the PLC to implement the rootkit. `CLIK` on the other hand focus on the control logic and does not modify the PLC firmware. Cyber-physical attack which is completely invisible to the control center of an ICS and they reverse engineered the central control loop mechanism of a widely deployed Allen Bradley 1769-L18ER-BB1B CompactLogix 5370 L1 Rev. B PLC. Harvey damages the underlying physical system, while providing the operators with the exact view of the system that they would expect to see following their issued control commands.We present a new infection hiding method based on a virtual PLC to overcome the limitations of the current methods, which typically require rigorous manual reverse engineering, which is a server program replying with valid response messages when it receives request messages from an engineering software.

Falliere *et al.* [9] explained how Stuxnet creators amassed a vast array of components to increase their chances of success. This includes zero-day exploits, a Windows rootkit, the first ever PLC rootkit, antivirus evasion techniques, complex process injection and hooking code, network infection routines, peer-to-peer updates, and a command and control interface [9]. Stuxnet's payload needs to be precompiled, which is not the case with our attack tool, `CLIK`. `CLIK` uses the decompiled code of a control logic, infect it and then compile it again to machine code at runtime.

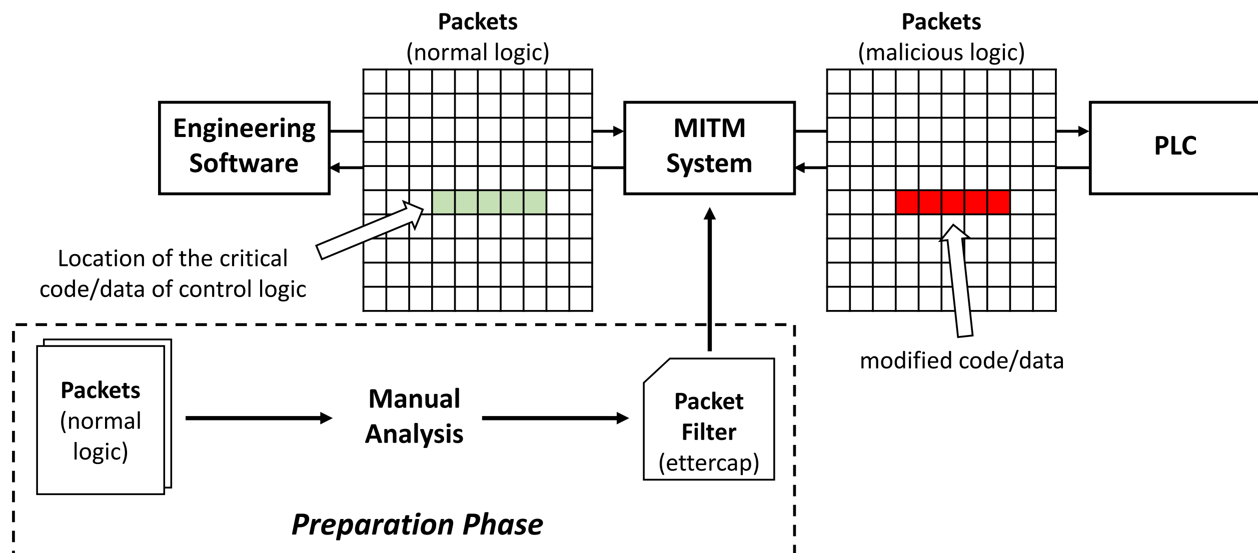


Figure 2.1: Infection hiding based on a packet filter

Senthivel *et al.* [15, 16] present three new attack scenarios referred to as Denial of Engineering

Operations (DEO) attacks where an attacker can interfere with the normal cycle of an engineering operation leading to a loss of situational awareness. The attacker can deceive the engineering software during attempts to retrieve the ladder logic program from a PLC by manipulating the ladder logic on the PLC such that the software is unable to process it while the PLC continues to successfully execute it. 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. DEO attacks [16], also uses Ettercap, which has limited support of writing comprehensive filters to cover complex control logic such as removing a series of malicious rungs in a ladder logic in network packets (refer to Figure 2.1). On the other hand, `CLIK` uses virtual PLC to engage an engineering software without crashing it.

Our decompiler `Eupheus`is the first developed for the Instruction List defined in IEC 61131-3 to transform the binary into its corresponding IL source code for Schneider Electric Modicon M221 and SoMachine-Basic engineering software. We further utilize rule-based approach to define criteria for automated malicious modification of the source code, and then compile the code to transform it into a binary that can run on the PLC. Then, the binary is transferred to the PLC to compromise the control logic. We present 3 different attack scenarios, to show that it is flexible enough to allow different types of attacks, since it can be configured to conjecture semantics of variables in control logic, depending on the degree of the adversary's knowledge about the system.

Chapter 3

Control Logic Infection Attack (CLIK)

Figure 3.1 shows a high-level overview of `CLIK`, the proposed control logic infection attack comprising of four phases: 1) Stealing the original control logic from a target PLC, 2) decompiling the stolen low-level (binary) representation of the control logic to its high-level source code, 3) infecting the source code via rule-based automated approach, followed by compiling the code to a binary representation (that can run on the PLC) and then, transferring the binary back to the PLC to infect the control logic, and finally 4) concealing the infected logic in the compromised PLC from an engineering software at the control center using a virtual PLC.

3.1 Phase I: Stealing the Original Control Logic from a Target PLC

The *first* phase involves gaining access to the control logic in a target PLC and retrieves the logic remotely over the network.

Subverting Security Measure This phase includes compromising any security measures that are supposed to protect a PLC from remote cyber attacks such as theft of control logic. The security measures include *integrity protection* of PLC firmware, configuration and control logic, *access control and firewall* to segregate PLC based on the access medium (such as the physical interface and communication protocol) and white-listing of IP addresses, and *authentication* (such as password) to restrict remote read/write access to PLC.

In Section 4.1, we will demonstrate an attack case of subverting the password authentication of a real PLC by exploiting a zero-day vulnerability that we found during the implementation of

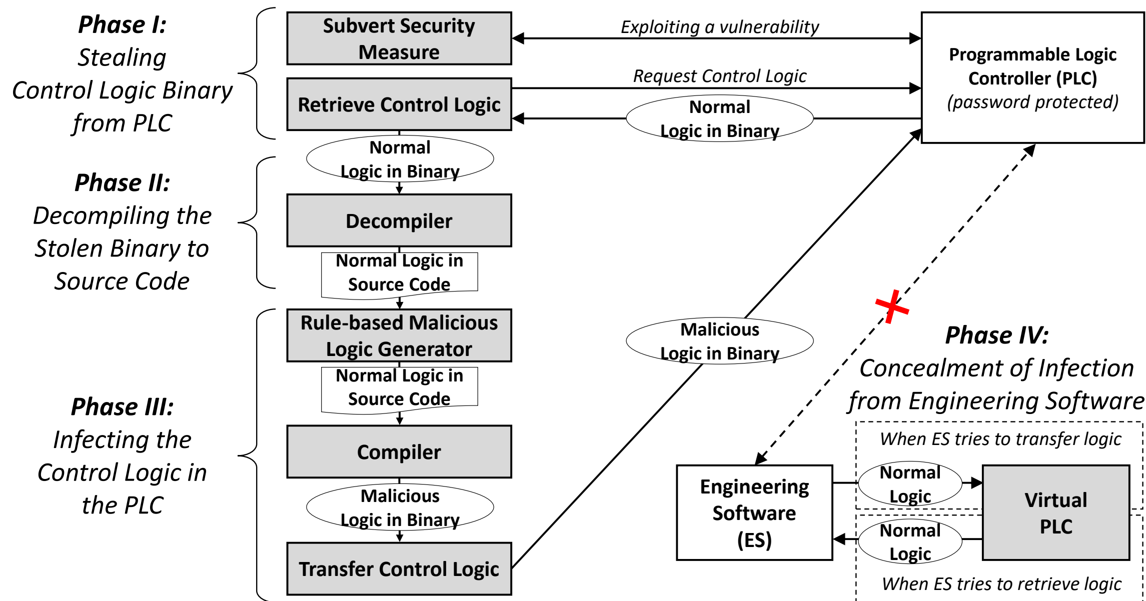


Figure 3.1: High-level overview of CLIK, an automated control logic infection attack

CLIK.

Retrieving Control Logic After security measures are compromised, CLIK retrieves the control logic from the PLC. It communicates with the PLC using the protocol supported by the PLC and then, requests the control logic. A control logic is typically divided into three parts: configuration (metadata) blocks, code blocks, and data blocks. Configuration blocks have the mapping addresses and sizes of other blocks. A code block is the machine code, which executes in PLCs. Data blocks contain values of variables used by a code block such as input, output, timer, and counter. In most cases, mapping addresses and sizes of code and data blocks vary, therefore CLIK first retrieves and parses the configuration blocks to get valid mapping addresses and sizes of other blocks.

3.2 Phase II: Decompiling the Stolen Binary to Source Code

The stolen control logic is in low-level (binary) format. The *second* phase decompiles the binary into its respective source code for automating the infection phase. The source code is written in one of the five high-level languages defined in IEC 61131-3 i.e., Instruction List, Ladder Logic, Sequential Function Charts, Function Block Diagram, and Structured Text. Figure 3.2 illustrates the decompilation process of CLIK, which takes the code block as an input and utilizes a database of

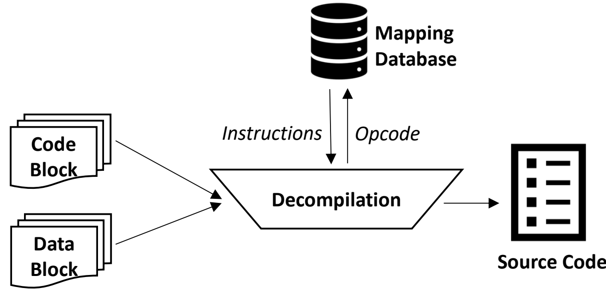


Figure 3.2: Decompilation of the Control Logic

opcode to instruction mapping for decompilation. Furthermore, `CLIK` takes into account the data block to obtain additional configuration parameters for the instructions. For instance, the timer instruction has the parameters of preset, time base and the type of timer (Timer On - TON, Timer Off - TOF, Pulse Timer - TP) in data block.

The decompiler maps each opcode in code block to the database to get the corresponding high level representation of the instruction in `Instruction List`.

3.3 Phase III: Infecting the Control Logic via Rule-based Approach

The *third* phase is the infection of the control logic that makes rule-based malicious modifications in the (decompiled) source code of the control logic, and then, compiles the modified infected code to a binary that can run on the PLC. Lastly, it transfers the binary to the PLC.

Rule-based Malicious Logic Generator `CLIK` employs a rule-based approach to serve two purposes: 1) identifying a target control logic and then, 2) infecting it automatically.

To *identify a target control logic*, `CLIK` leverages the semantics of the (decompiled) source code and estimated range of critical variables of a target physical process. In many cases, the source code contains sufficient semantics of the underlying physical process to generate rules. `CLIK` for instance, can look for special instructions in the decompiled code to infer meaning of specific variables such as set-points. These instructions include PID instructions of Allen-Bradley RSLogix 500, and drive blocks of Schneider Electric SoMachine-Basic. Furthermore, `CLIK` can look for the data to identify estimated range of critical variables representing a physical process. For instance,

similar to Stuxnet, `CLIK` can observe the data that range between 807 Hz and 1,210 Hz to identify variable frequency drives of centrifuges.

To *infect the identified control logic*, `CLIK` utilizes preconfigured infection rules including the following: replacing input or output bits with memory bits (to disturb normal update on output pins), replacing operators in equations, modifying set-points, modifying control flow determinants (to influence a decision at a conditional branch of control logic), insert/delete instructions/rungs, etc. For example, similar to Stuxnet, an infection rule can be configured to insert a new rung that manipulates the set-point of motor speed periodically from 1,410 Hz to 2 Hz to 1,064 Hz leading to sabotage the centrifuges controlled by the target PLC.

In this phase, we use the resultant decompiled code from network and modify the Instruction List as a string array and replace or add new rungs and instructions based on the rules and the type of attack selected.

We present four heuristic rules that can be implemented while changing the Instruction List: rule 1) replacing input or output bits with memory bits, rule 2) modifying analog control flow determinant, rule 3) modifying set-points, and rule 4) replacing operators.

The rule 1 can be generally applied for modifying digital values in control logic. By disabling input and output bits of a control logic, it performs a denial-of-service attack in effect. On the other hand, rule 2, 3, and 4 focus on modifying analog values in control logic. By examining analog values, `CLIK` could be configured to find a target variable in control logic to conduct more sophisticated modification. If attackers have some estimated range of the target variable, `CLIK` can try to find the target variable based on the estimated value range. A target value estimation can be done in two ways: observation and general feature.

If the attacker is able to observe the features of the target physical process, for example, the time period of a traffic light system, that information can be used to configure the malicious logic generator to find a specific timer preset. Otherwise, the target physical system could have some general feature. For example, a centrifuge for enriching Uranium-235 is required to spin at 50,000 $\sim$ 70,000 rpm. Therefore, attackers who want to sabotage a nuclear facility can use this information to configure the malicious logic generator to find and modify the specific variable related to the target variable-frequency drive of a gas centrifuge.

Infection rule 1) Replacing input or output bits with memory bits The infection may make the PLC unresponsive by replacing memory bits with input or output instructions. This approach mainly focuses on the digital values in control logic, which can only result into a static attack with same or similar kind of results when a new rung is appended to the original control logic.

Infection rule 2) Modifying analog control flow determinant We define that a variable X is a control flow determinant if X influences a decision at a conditional branch of control logic. For example, analog control flow determinants include variables of function blocks such as timer preset and counter preset. The modification is done by following steps: 1) finds all analog control flow determinants except input/output variables in control logic, 2) if there is configured range of target variables, finds matching variables among the control flow determinants and modifies them with configured values, 3) if there is no configured value or fails to find the matching variable, randomly selects an analog control flow determinant and modifies it with a random value.

Infection rule 3) Modifying set-points A set-point is a desired value for a process value of systems like a motor frequency of a gas centrifuge. In most of control logic program, a set-point of underlying physical process is represented using special function blocks or instructions provided by an engineering software. For example, Schneider Electric SoMachine-Basic provides drive block which allow drive devices to be controlled by a PLC. Allen-Bradley's RSLoigx 500 supports proportional integral derivative instructions which control a process variable to be at a desired set-point using a closed process loop. The modification is done by following steps: 1) finds those function blocks or instructions which contain a set-points, 2) if there is configured range of target variables, finds matching variables among the set-points and modifies them with configured values, 3) if there is no configured target value or fails to find the matching variable, randomly selects a set-point and modifies it with a random value.

Infection rule 4) Replacing operators in operational blocks Changing the operational block equation can change the flow of the entire control logic. This modification can be accomplished by replacing the operators with an opposite operator (like ≔ can be replaced with <>) which makes the control flow to act exactly opposite than it was originally intended to.

Compiler After the decompiled code block is infected, it is compiled to a binary that can run on the target PLC. The compiler uses the same database that the decompiler used for conversion but the other way around. It searches for the instruction opcode for a high level instruction in the database and replaces the former with the latter.

Transfer of the Control Logic to the PLC The infected control logic is transferred to the target PLC using the write-requests of the PLC protocol. The modified (infected) control logic blocks should be mapped to the address of the original blocks to ensure the stable transition of the PLC to the infected logic.

If the target PLC accepts the write requests of control logic blocks at a specific state of the protocol implementation, it needs to transit to the valid state before sending write requests. In addition, if the PLC requires a checksum for control logic, the checksum for the infected control logic should be calculated.

3.4 Phase IV: Concealing the Malicious PLC Control Logic from Engineering Software

When the control logic in a target PLC is infected, `CLIK` hides the infection from the engineering software to sustain the operation of the infected logic. Existing approaches i.e., Stuxnet [9] and DEO attacks [16] intercept and modify the packets between the engineering software and PLC to hide the infection. Stuxnet [9] intercepts control logic packets by replacing `s7otbxdx.dll` of `SIMATIC STEP 7` engineering software that handles communication with the PLC and then, masks the infection. Furthermore, DEO attacks [16] utilize a packet filter of Ettercap [2] and ARP poisoning to intercept the packets and then, removes the malicious changes in the control logic before forwarding them to the engineering software.

The current approaches hide the infected control logic successfully. However, they have several limitations. Stuxnet compromises the integrity of a dynamic link library, which demands advanced attack capabilities to deceive integrity checking services. DEO attacks [16], on the other hand, uses Ettercap, which has limited support of writing comprehensive filters to cover complex control logic such as removing a series of malicious rungs in a ladder logic in network packets (refer to

Figure 2.1). Also, it leaves several duplicate packets in the network trace due to packet forwarding, which a network operator can quickly notice by using network monitoring tools such as Wireshark. Moreover, since the packet filter is generated based on one specific normal control logic, it only works for that control logic. If the engineering software downloads other control logic to the target PLC, their packet filter could not correctly locate the critical parts of the control logic.

Virtual PLC We present a new infection hiding method based on a virtual PLC that overcomes the limitations of the existing approaches. The goal of the virtual PLC is to achieve stealthiness by avoiding significant perturbation in the environment including installing malicious DLL or generating duplicate traffic.

The virtual PLC utilizes the captured network traffic of the original control logic obtained in phase 1 of `CLIK` to engage engineering software. It is a server program that intercepts the request messages from an engineering software and then, replies with valid response messages using the captured traffic. In other words, the virtual PLC mimics the behavior of an uninfected real PLC from engineering software's viewpoint.

A Systematic Approach of Building a Virtual PLC. Building a virtual PLC requires reverse engineering especially when a target PLC uses a proprietary protocol. In this paper, we present a systematic approach to build a virtual PLC. Figure 3.3 presents a systematic approach to build a virtual PLC. The virtual PLC faces two main challenges. 1) when engineering software sends a request message, virtual PLC has to identify the corresponding response messages in the captured traffic, and 2) also adjust the dynamic fields in the response messages whose values vary within and across different sessions. To solve these challenges, the virtual PLC develops *communication template* and *essential protocol format* in two stages for a target PLC. It executes both stages offline during the preparation of the `CLIK` attack.

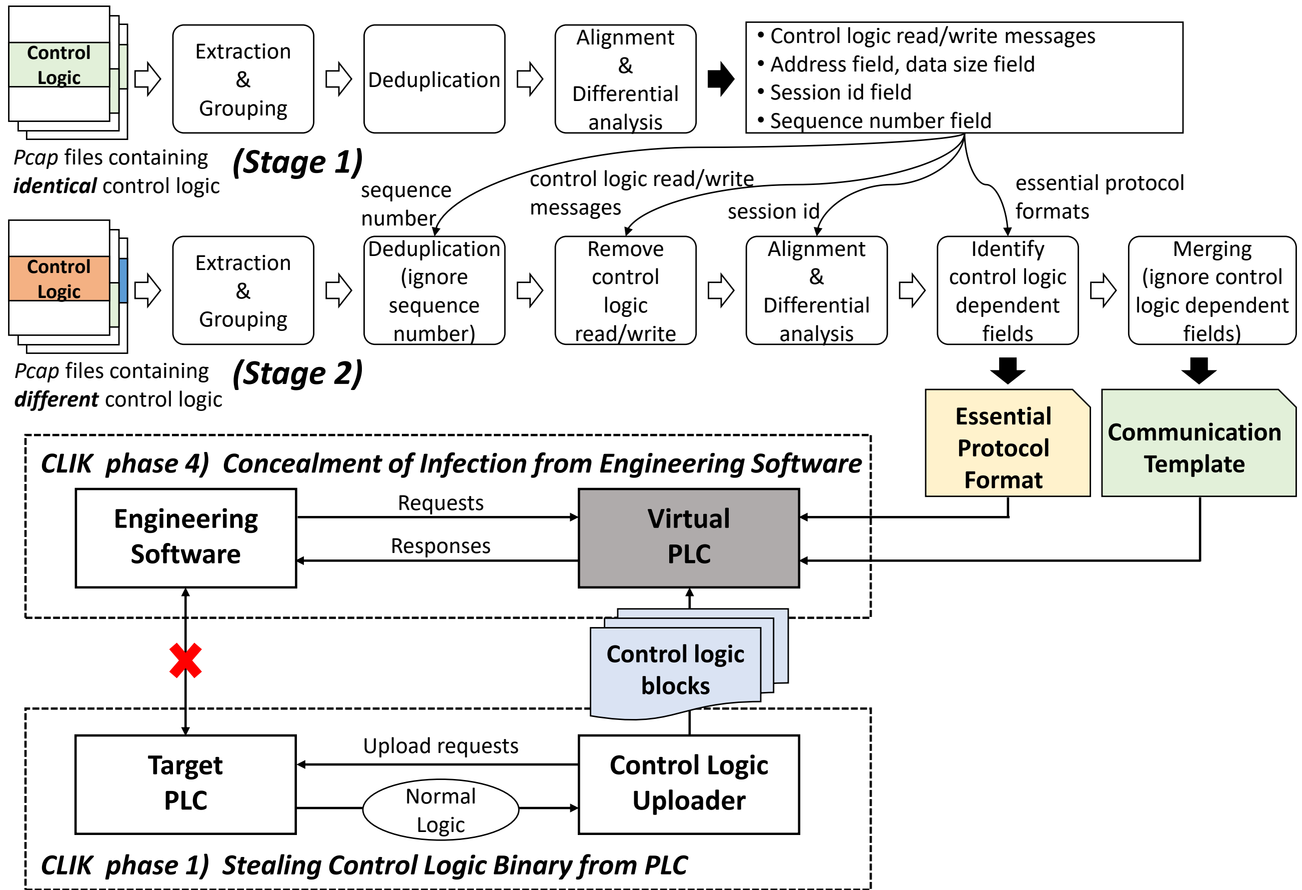


Figure 3.3: A systematic approach to build a virtual PLC

We build virtual PLC based on our insight that the regular traffic (excluding messages containing control logic in their payload) between engineering software and PLC consist of a manageable number of different messages that can be represented as a *communication template*. If a virtual PLC receives a request message, it looks up the template to find a matching request and replies with the corresponding response message. This approach is black-box that does not require the complete semantic knowledge of request/response message content, thereby, saving time and effort for reverse engineering every aspect of the protocol.

Virtual PLC, however, has to take into account some dynamic fields, of which values may change within a session (e.g., sequence number), between sessions (e.g., session id), and for control logic (e.g., control logic checksum). Also, since the communication template does not include messages containing control logic, the virtual PLC has to understand how to generate valid control logic read/write response messages. To this end, the virtual PLC needs to understand these essential

parts of the protocol format, refer to as *essential protocol formats.*This section further describes a two-stage process of deriving a *communication template* from network-traffic-captures and building an *essential protocol format.*

Deriving Essential Protocol Format. Stage 1 collects multiple network packet captures of the same control logic and then, processes each packet capture as follows: protocol messages are extracted and grouped as request and response pair. For instance, some proprietary PLC protocols are encapsulated by the Modbus/TCP protocol which is a de facto standard communication protocol widely used in industrial control systems. Since a Modbus request and its corresponding Modbus response have the same transaction identifier number, encapsulated PLC protocol messages can be grouped based on the transaction identifier of the Modbus/TCP protocol. After pairing, duplicate pairs are eliminated. Duplicated messages can exist if there is periodical status check between a PLC and its engineering software. This process is repeated for all packet captures.

After that, messages in packet-captures are aligned and differences are analyzed. It is possible to recognize a large chunk of aligned message, since the messages contain same control logic, which leads to find essential format of a proprietary protocol. We will describe how the alignment facilitates format recognition for a proprietary PLC protocol in Section 4.6. The essential protocol format includes dynamic fields such as session id and sequence number, and control logic read/write message formats.

Building a Communication Template. Stage 2 is similar to stage 1 except that it uses different control logic programs to collect multiple network packet captures. It processes each network capture by extracting and grouping the messages in request/response pairs, followed by eliminating duplicate pairs of the messages. If a sequence number field is recognized in stage 1, it is ignored in the deduplication step. The request and response messages involving control logic read/write are removed based on the essential protocol format inferred in stage 1.

The remaining messages of the packet captures are aligned, and differences between them are analyzed. At this step, a session id field is ignored. The aligned message chunks reveal control-logic dependent fields (since other dynamic fields are ignored in the previous steps) including checksum fields. These fields, as well as the previously derived format in stage 1, compose *essential protocol format.* Lastly, merging ignores control-logic dependent fields to make a communication template consisting of the unique request and response pairs.

During the attack, virtual PLC finds the corresponding messages to a request message using the *communication template* and then, adjusts the dynamic fields using the *essential protocol format* before sending the messages to engineering software. If the request message is not in the template, it means it is a control logic read/write request message. The virtual PLC generates a valid response message according to the current control logic obtained in phase 1 of `CLIK`.

Chapter 4

Real-world Implementation of CLIK

We implement CLIK on a real PLC, Schneider Electric Modicon M221, and its vendor-supplied engineering software (SoMachine-Basic). CLIK is developed in Python and consists of five modules: 1) a password attack module, 2) a control logic upload & download module, 3) IL decompiler & compiler, 4) a malicious logic generator, and 5) a virtual M221 PLC. Since the M221 PLC uses a proprietary protocol and the format of the control logic blocks were unknown, we had to do some reverse engineering. The detailed efforts of our reverse engineering works are not described here for space reason.

4.1 Subverting PLC Password Protection by Exploiting a Zero-day Vulnerability

Since the latest version of the M221 PLC is protected by a password authentication mechanism, an unauthorized user is not allowed to read the control logic of the PLC. For example, one can not upload the control logic of the M221 PLC without the correct password if the password protection option is enabled. We have found a zero-day vulnerability in the authentication mechanism and further develop a proof-of-concept exploit to subvert the password protection. We confirmed that the latest version of firmware (v1.5.1.0 and v1.6.0.1) and SoMachine-Basic (v1.5 and v1.6) are impacted by this vulnerability, and reported this to the vendor.

Figure 4.1 describes the authentication process between an M221 PLC and SoMachine-Basic. First, the engineering software requests a one-byte random mask (*mask1*) from the PLC. After receiving (*mask1*), the engineering software sends one-byte random mask (*mask2*) to PLC along with the masked value of the password hash for authentication. The masked value is computed by XORing the original password hash (SHA-256) against both *mask1* and *mask2*. When the PLC receives the masked hash and *mask2*, it computes its masked-hash using the same XOR operation

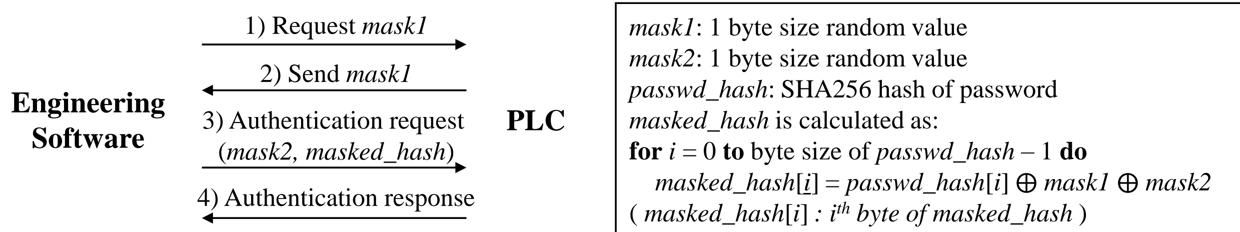


Figure 4.1: Normal password authentication between the Schneider Electric Modicon M221 PLC and SoMachine-Basic engineering software

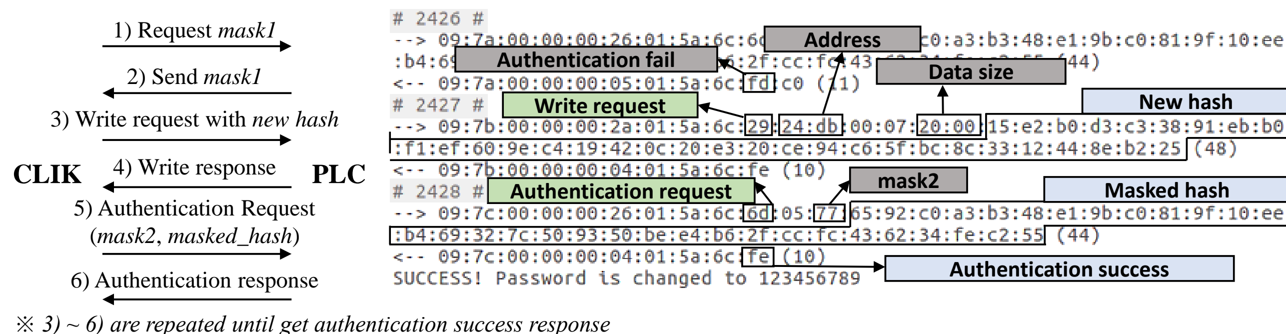


Figure 4.2: Password authentication while exploit the vulnerability in the Schneider Electric Modicon M221 PLC by the `CLIK`

initially performed by the engineering software. If both masked-hashes are identical, the PLC grants access permissions to the engineering software.

The M221 PLC does not allow reading control logic remotely before authentication. However, writing to the PLC is still allowed, which is an exploitable vulnerability. We demonstrate that an attacker can reset the password by overwriting the original password hash with its own. However, the challenge for the attacker is to identify the correct location of the hash code in the PLC to overwrite.

Figure 4.3 shows the address space layout of the M221 PLC. We find out that the password hash is always preceded by a variable size `zip` file containing metadata of the control logic. We also discover that the zip file is mapped at the fixed address (`0xd000`). Furthermore, the code block of the control logic, which is a machine code of Renesas RX630 microcontroller [14], is always mapped at an arbitrary location starting from `0xe000`, referred to as *random gap*. Thus, we compute that there is an *unused area* between the end of the password hash and the address `0xe000`.

Although an attacker is not allowed to read control logic from the PLC without knowing the

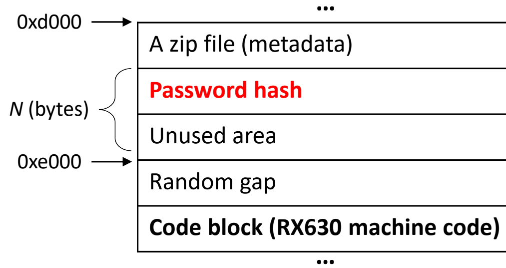


Figure 4.3: Address space of the M221 PLC

valid password, the attacker still can write values to the PLC. Therefore, if an attacker knows the address where the authentication key (e.g. password hash) is stored, the attacker can pass the authentication by overwriting the original authentication key with attacker's key.

The address is represented by a two-byte field in read/write messages of the M221 PLC protocol. The M221 PLC uses a password hash as authentication key, and the password hash is always preceded by a variable size zip file containing metadata of the control logic. We also found out that the zip file is mapped at fix address (`0xd000`) and the code block of the control logic, which is a machine code of Renesas RX630 microcontroller [14], is always mapped at arbitrary address above `0xe000`. Therefore, there is unused area between the end of the password hash and the address `0xe000`.

To identify the correct location of the hash code to overwrite with new password hash, the attacker can compute negative or positive offsets from a known reference point to password hash. Unfortunately, the offsets are not consistent because of the following reasons. To compute the *positive* offset, we know the starting address of the zip file. However, the size of the file varies and cannot be found during the attack. To compute the *negative* offset, we know that the code block of control logic starts from any random location after `0xe000`. However, the random location is not apparent making the offset value unpredictable.

The other approach is to fill the memory between the `zip` file and `0xe000` with the attacker's password hash. However, this approach is not reliable. It can start overwriting the memory from `0xe000` to password hash, but cannot precisely determine the end of the zip file since the file size varies, which may cause exceeding the hash location and overwriting the zip file contents.

Furthermore, the size of the SHA-256 hash is 32 (0x20) bytes. The original hash must be located by the multiple of 32-byte from `0xe000` to ensure that the new hash completely overwrites the old

hash.

Figure 4.2 shows the overview of the password authentication while exploit the vulnerability. Algorithm 1 describes the exploit method that takes advantage of the vulnerability and resets the original password of the M221 PLC with an attacker's password. It assumes that the password hash is located anywhere starting from `0xe000` to a negative offset. Since the size of the SHA-256 hash is 32 bytes (0x20), it first overwrites the address `0xdfe0` $\sim$ `0xe000` with the new password hash (line 10 in Algorithm 1) and then, sends an authentication request to the target PLC using *mask2* and *maskedHash* (line 11). If the authentication fails, it iteratively performs the same steps. However, every iteration overwrites the hash value to the memory location, which is one-byte negative offset from the last address. For instance, the next address after the first failed authentication is `0xdfdf` $\sim$ `0xdfff`. At some point, an iteration overwrites the original hash code completely with the correct alignment, which resets the password and authenticates successfully.

Since the size of the SHA-256 hash is 32 (0x20) bytes, it first overwrites the address `0xdfe0` $\sim$ `0xe000` with the new password hash (line 10). Then, send an authentication request to the target PLC using *mask2* and *maskedHash* (line 11). If the authentication fails it overwrites the next address `0xdfdf` $\sim$ `0xdfff` and send an authentication request again. In this manner, it iteratively overwrites the address space until an authentication success message is received.

It first calculate *newHash* (line 1) and get *mask1* from the PLC (line 2). Then, *mask2* is set with an arbitrary value (line 3). *maskedHash* is calculated from *newHash*, *mask1*, and *mask2* (line 6-8). Then it iteratively overwrite *targetAddr* with the *maskedHash* as follow (line 9-14). When the size of the password hash is 32 (0x20) bytes, an attacker first overwrite the address `0xdfe0` $\sim$ `0xe000` with the attacker's password hash, then send an authentication request to the target PLC using the attacker's password hash. If the authentication fails the attacker overwrite the address (`0xdfe0 -` 1) $\sim$ (`0xe000 -` 1) and send an authentication request again. In this way, the attacker iteratively overwrite the address (`0xdfe0 -` 2) $\sim$ (`0xe000 -` 2), (`0xdfe0 -` 3) $\sim$ (`0xe000 -` 3), ..., (`0xdfe0 -` N) $\sim$ (`0xe000 -` N), until authentication succeeds.

Algorithm 1 Pseudocode for password reset attack

Input: New password
Result: Reset password with the new password

```
1:  newHash ← sha-256 hash of the new password
2:  mask1 ← Request mask1 from the PLC
3:  mask2 ← A random number between 0 and 255
4:  targetAddr ← 0xdfe0
5:  hashSize ← 32
6:  for i = 0 to hashSize − 1 do
7:      maskedHash[i] ← newHash[i] ⊕ mask1 ⊕ mask2
8:  end for
9:  while res = False do
10:     Send a write request (addr:targetAddr, size:hashSize, data:newHash) to PLC
11:     Send an authentication request (mask2, maskedHash) to PLC
12:     res ← Received result of the authentication request
13:     targetAddr ← targetAddr − 1
14: end while
```

4.2 Control Logic Uploader & Downloader

The *Uploader* and *Downloader* retrieves the control logic from and transfer it to a PLC respectively. In M221 PLC, the control logic consists of six blocks. We refer to them as `conf1`, `conf2`, `code`, `data1`, `data2`, and `zipHash` since the protocol is proprietary and its specification is not publicly available. The `conf1` and `conf2` blocks have information of other blocks. The `code` block is the compiled version of control logic in RX630 machine code. The `zipHash` block consists of a zip file (metadata of control logic) and password hash (if password protection option is enabled). The `data1` block contains the values of the variables used in control logic. The `data2` block has information about the `data1` block.

The *Uploader* reads all six blocks of the control logic from the M221 PLC. It first obtains the address and size of each block and then, sends read messages to the PLC to retrieve the entire logic. Figure 4.4 illustrates the process of getting the address and size of the blocks. We find that the `conf1` block is always mapped at a fixed address (`0xfed4`) and has the fixed size of 300 bytes. The `conf1` block is then used to derive the size of the `zipHash` block, and the size and address of the `conf2` block. The `conf2` block is retrieved from the PLC and then, used to derive the size of the `data1`, and `data2` blocks, and the address and size of the `code` block. After obtaining the addresses and sizes of the control-logic blocks, the *Uploader* sends read-requests to the PLC to retrieve them.

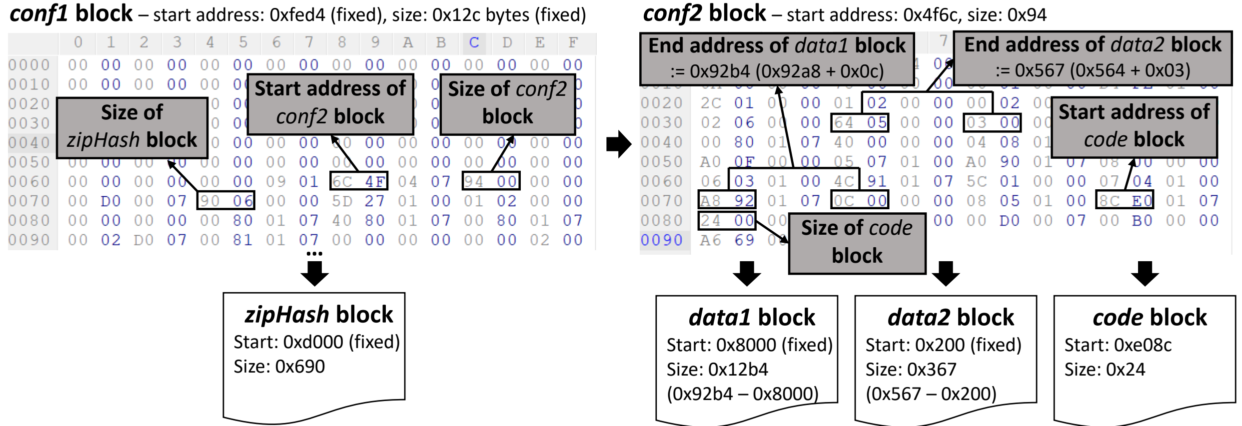


Figure 4.4: Start address and size of M221 control logic blocks

The *Downloader* sends write-requests to the PLC with the corresponding addresses and size of the control-logic blocks. During the `CLIK` attack, the *Malicious Logic Generator* modifies either `data1` or `code` block, or both. The *Downloader* sends write-requests containing modified-blocks with their original addresses and current sizes that may be different from original sizes after modifications.

4.3 IL Decompiler & Compiler

SoMachine-Basic (an engineering software) supports two programming languages, ladder logic, and IL. The ladder logic consists of rungs and each rung has instructions. The SoMachine can interchange the instructions between the two languages. When SoMachine compiles a control logic program, it converts the program into RX630 assembly code.

IL Decompiler We implement the decompiler `Eupheus` that takes the RX630 code as input and decompiles it into IL source code. We choose IL over ladder logic because IL is text-based and easier to manipulate. Ladder logic, on the other hand, is a graphical language where each instruction is represented as a graphical symbol and the instructions are grouped into rungs.

The decompiler `Eupheus` has two main components: First, the *database* for mapping the opcodes to their corresponding IL instructions (refer to Figure 4.7). Currently, the database consists of 4079 mappings for 21 types of different instructions including input and output, relative branch, control block, and operational block. Second, the *mapper* program, which utilizes the `data` and `config`

blocks in the M221 control logic and the mapping database to process the `code` block. In the first step of decomilation mapper divides the control logic into individual rungs, using two criteria, a rung can be separated when an output instruction is followed by an input instruction or if the rung has a control block instruction like `Timer` or `Controller` it ends with the `END_BLK` instruction which is considered as a delimiter in the database. when the rungs are separated and stored into an array, the *mapper* proceeds to the next step: replacing the binary with appropriate instructions. The *mapper* finds the RX630 instructions of the `code` block in the database of opcodes to obtain their corresponding IL instructions. We notice that RX630 program maintains the essential constructs of both languages, IL and ladder logic to facilitate the decompilation by SoMachine. The *mapper* therefore, first identifies the rungs in RX630 program and then, processes each rung to identify IL instructions.

The Table 4.1 shows an example of RX630 control logic that is mapped to IL instructions. The rungs are separated when an output instruction opcode is preceded by the instruction opcode of an input or a control block instruction such as `Timer` or `Counter`. In the latter case, the rung ends with `END_BLK` instruction denoted as `7F 1A 11` in RX630 control logic, The Figure 4.8 shows an example of control logic that is using timer. After the rungs are separated, `Eupheus` decompiles each rung at a time by identifying the instructions in the database and replacing the RX630 instruction with equivalent IL instruction. The bitwise `AND` and `OR` are denoted in control logic as `0x23` and `0x22` respectively, followed by the byte that indicates the total number of bytes of the next instruction. For example, `7C 0C 23 04 7C 1C` represents that the two instructions `7C 0C` and `7C 1C` are in series connection and the `04` shows that the next instruction has two bytes.

The control logic sometimes contain `Operational Blocks`, equations using memory floats or memory words, which are used as control flow determinants for the physical process. In such cases binary uses the opcode `7F 1A` which means Jump to Sub-Routine in RX630. The mapper database is not enough to decompile in this situation. So we implemented a separate program to decompile the operation block equations. There is a specific code for each arithmetic operation in the binary. we stored each operation so that when we come across the operation code, the operational block decompilation program can be used. The binary has a specific structure for the equation as shown in the Figure 4.5.

When the equation is complicated then the binary is divided into multiple simple equations

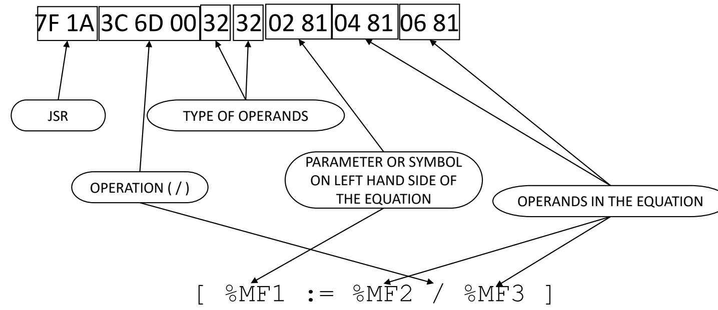


Figure 4.5: Simple Operation Block

with temp variables. The Figure 4.6 shows an example of complex equation breakdown in the control logic binary and the usage of `TEMP` variable to connect each simple equation and form the resultant complex equation. In the example given below the operand type `32` means the variable and the type `29` means constant. Each operation have different code in the binary with one to one connection.

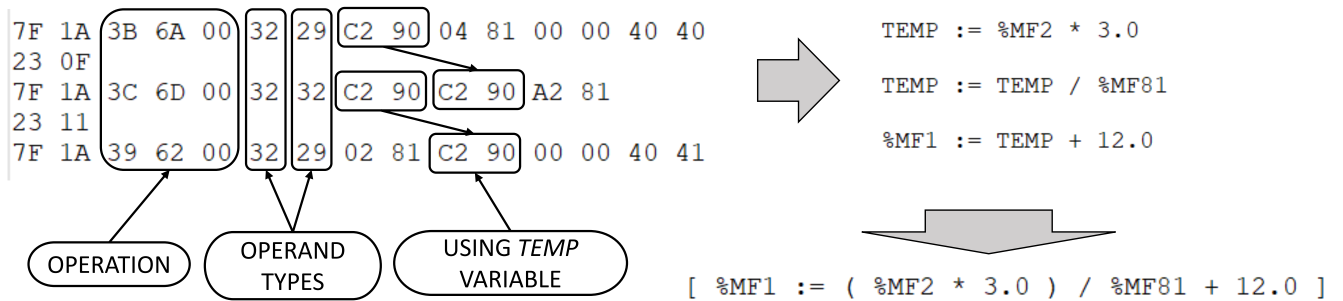


Figure 4.6: Complex Operation Block

The mapper program finds the operation block binary and sends it to the `Operational Block Decompiler`. the programs checks the binary and determines the complexity of the equation. The simple equation can be decompiled by getting the operator from the database and decompiling the constant and the variable. However when the equation is complicated like in the example given in Figure 4.6 we have to determine the temp variable initialization, utilization and the assignment of the `TEMP` variable. in the first step we decompile individual equations with the temp variable. Next we replace the temp variable with the equation assigned to it to form the final equation. The tricky part is as given in the figure if the temp variable is reassigned, that is if the temp variable is utilized to make another equation to reassign. We use the top-down approach to replace the

equations step by step.

Mapping Database The database maintains the opcodes of RX630 instructions and their corresponding IL instructions along with the category of the instruction such as inputs, outputs and relative branch instructions (`AND`, `OR`, etc.). The logic also has timer specifications such as `23 00` indicates that the timer type is `TON` and the preset is `1.00 second`.

includes Instructions table with the opcodes with their corresponding instructions and the category of the instruction (inputs, outputs and relative branch instructions like `AND` and `OR`). MetaData table includes timer specifications like `23 00` indicates that the timer type is `TON` and the preset is `1s`

The database stores each instruction opcode and the equivalent instruction in a table called `Instructions`. This table includes Input instructions, output instructions, relative branch instructions and block instructions which are constructed based on RX630 microcontroller assembly language.

However, The conditional blocks or the operational blocks in the Instruction list may not be completely mapped to the assembly code of RX630, as they use Jump to Sub Routine(7f 1a 10) to represent the `Block` instruction and after that the Hex code represents each input and output differently even then we were able to see some pattern to map the Instruction list and the code from network packet. Also the Operational blocks that use Memory words/floats are also represented with control logic that starts with (7f 1a) but the next few bytes of code is not as explained in the microcontroller assembly language.

The database includes Instructions table with the opcodes with their corresponding instructions and the category of the instruction (inputs, outputs and relative branch instructions like `AND` and `OR`). MetaData table includes timer specifications like `23 00` indicates that the timer type is `TON` and the preset is `1s`

Mapping Program The mapper program processes the M221 control logic by first dividing it into separate rungs. Each rung may have opcodes of input, output and conditional block instructions that are searched in the database to find their equivalent IL instructions.

In the cases of operational block, the mapping database is not enough to decompile the binary

of control logic.

IL Compiler The compiler is the reverse process of the decompiler `Eupheus`, which uses the same database to get the equivalent RX630 opcode of every instruction in Instruction List.

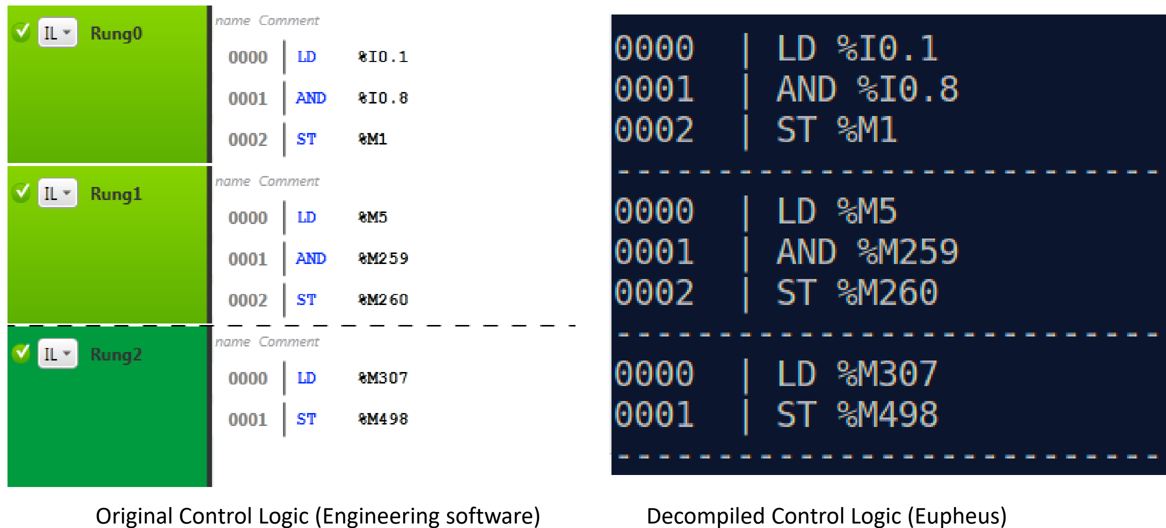


Figure 4.7: Mapping RX630 Opcodes with IL Instructions

4.4 Mapping RX630 Opcodes with IL Instructions

`CLIK` employs rule-based approach to inject malicious logic in the decompiled IL code of RX630 control logic automatically. The decompilation ensures that `CLIK` understands the RX630 control logic correctly since the opcode of each instruction is of different size; a longer opcode of an instruction may contain one or more smaller instruction opcodes. Furthermore, the original control logic is written in high-level language such as IL. Thus, it is easier to identify malicious logic for an IL control logic code than RX630 machine code.

4.5 Malicious Logic Generator

We used the decompiled Instruction List to modify or insert rungs to perform malicious activities when the code is executed by the PLC. The reason behind this is the opcode of each instruction is in different length, the longer instruction opcode the more chances that it might contain one or more smaller instruction opcodes that mean absolutely different when converted into Instruction List,

Table 4.1: Instruction List representation and corresponding machine code description

IL	Hex	Assembly Language	Description	Ref. (the manual [14])
Rung 0				
LD %I0.1	7c 1c	BTST 1(imm), R12	Z = ∼((src2 >>(src & 31)) & 1); C = ((src2 >>(src & 31)) & 1);	p.194 (3) BTST src, src2
AND %I0.8	23 04	BCnd.B 4(pcdsp) #cd: BNC(C == 0)	if (Cnd) PC = PC + src; PC += 4: if I0.1 is 0, then it skip the next instruction	p.185 (2) BCnd.B src
	7c 8c	BTST 8(imm), R12	Z = ∼((src2 >>(src & 31)) & 1); C = ((src2 >>(src & 31)) & 1);	p.194 (3) BTST src, src2

ST
%M1
	fc e6 72 00 00	BMCnd
		1(imm), [R7].B
		#cd: BMC
		(C==1)
		#dsp: 0x0000
			if (Cnd)
			dest —= (1 <<(src & 7));
			else
			dest &= ∼(1 <<(src & 7));
				p.187
				(1)
				BMCnd src, dest

Rung 1				
LD %M5	f6 75 00 00	BTST 5(imm), [R7].B #dsp: 0x0000	Z = ∼((src2 >>(src & 7)) & 1); C = ((src2 >>(src & 7)) & 1);	p.194 (1) BTST src, src2
AND %M259	23 06	BCnd.B 6(pcdsp) #cd: BNC(C == 0)	if (Cnd) PC = PC + src; ※if M5 is 0, then it skip the next instruction	p.185 (2) BCnd.B src
	f6 73 20 00	BTST 3(imm), [R7].B #dsp: 0x0020	※Target Memory addr = R7 + 0x20 Target bit offset is 259 (0x20 * 8 + 3)	p.194 (1) BTST src, src2
ST %M260	fc f2 72 20 00	BMCnd 4(imm), [R7].B #cd: BMC (C==1) #dsp: 0x0020	※ Target Memory addr = R7 + 0x20 Target bit offset is 260 (0x20 * 8 + 4)	p.187 (1) BMCnd src, dest
Rung 2				
LD %M307	f6 73 26 00	BTST 3(imm), [R7].B #dsp: 0x0026	※Target Memory addr = R7 + 0x26 Target bit offset is 307 (0x26 * 8 + 3)	p.194 (1) BTST src, src2
ST %M498	fc ea 72 3e 00	BMCnd 2(imm), [R7].B #cd: BMC (C==1) #dsp: 0x003e	※Target Memory addr = R7 + 0x3e Target bit offset is 498 (0x3e * 8 + 2)	p.187 (1) BMCnd src, dest
	02	RTS	PC = *SP; SP = SP + 4; ※This instruction returns the flow of execution from a subroutine	p.240 (1) RTS

Figure 4.9 shows some examples of such cases, where inserting or changing the code is complicated.

To simplify we use the decompiled instruction list so that the change can be significant.

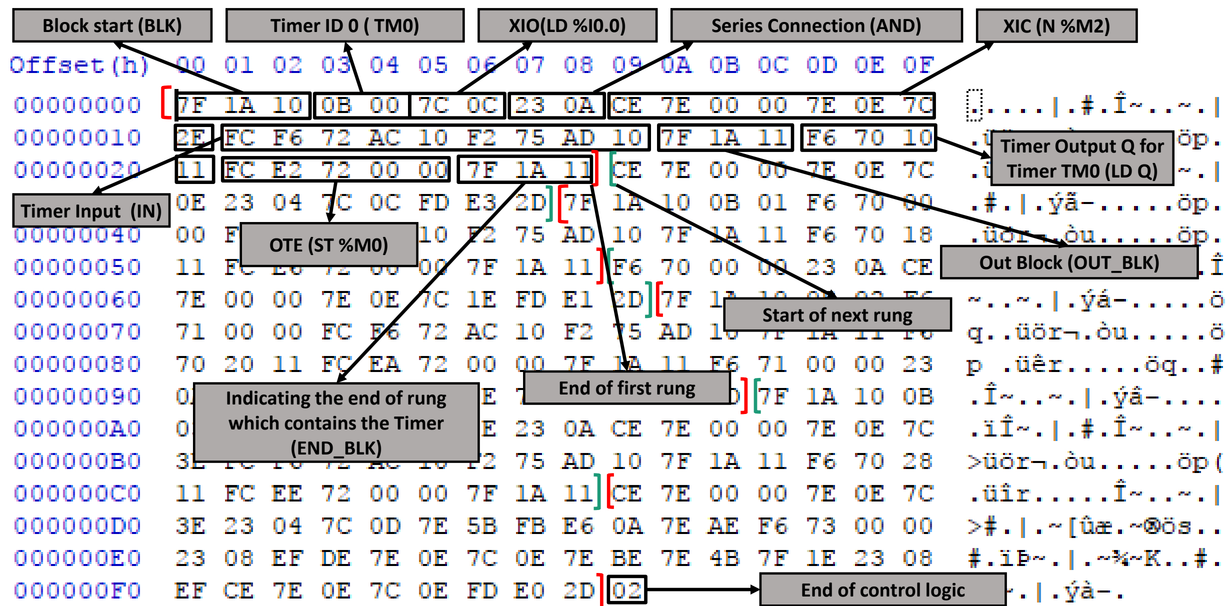


Figure 4.8: Mapping opcode with Instructions

Rules We define five heuristic rules to modify the IL code; more rules can be included to improve the malicious logic generator: *Rule-1:* If input/output bits are found then, replace them with memory bits. *Rule-2:* If a configured range of target variable matched then, modify its determinant. *Rule-3:* If a set-point is less than a ¡certain value¿ then, modify the set-point. *Rule-4:* If an operational block has an equation then, replace an operator in the equation. For instance, the operator := can be replaced with <>. *Rule-5:* Insert a new rung at the end of the logic with an energizer output of a target actuator to override the output with the attacker's desired value.

The rule 1 can be generally applied for modifying digital values in control logic. By disabling input and output bits of a control logic, it performs a denial-of-service attack in effect. On the other hand, rule 2, 3, and 4 focus on modifying analog values in control logic. By examining analog values, `CLIK` could be configured to find a target variable in control logic to conduct more sophisticated modification. If attackers have some estimated range of the target variable, `CLIK` can try to find the target variable based on the estimated value range. A target value estimation can be done in two ways: observation and general feature.

First, attackers might be able to observe the features of the target physical process. For example, attackers can observe time period of a traffic light system and use that information to configure the malicious logic generator to find a specific timer preset. Otherwise, the target physical system could have some general feature. For example, a centrifuge for enriching Uranium-235 is required to

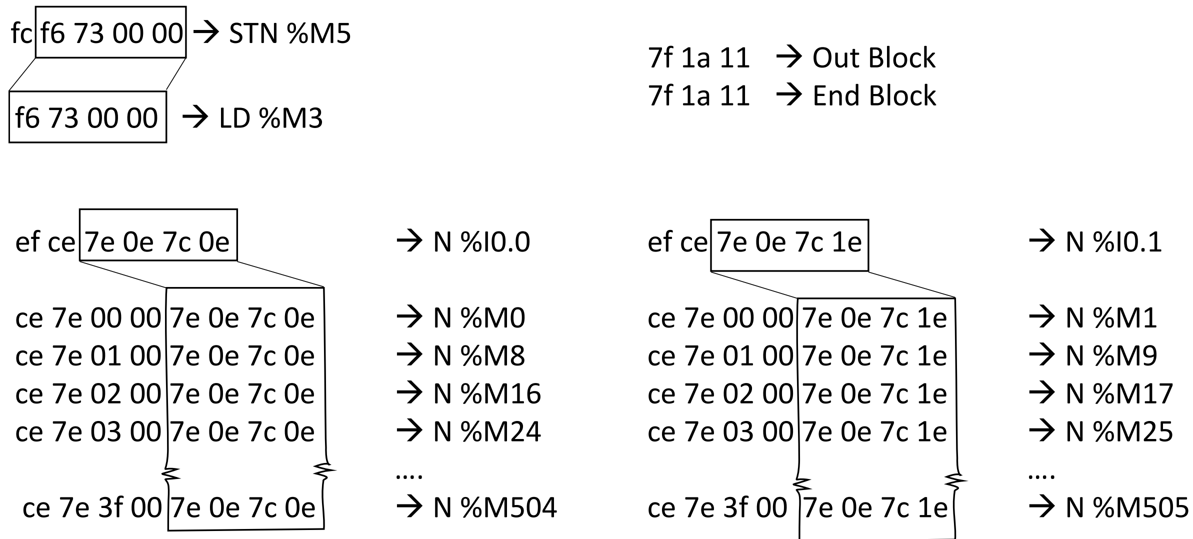


Figure 4.9: Similar Opcode Examples

spin at 50,000 $\sim$ 70,000 rpm. Therefore, attackers who want to sabotage a nuclear facility can use this information to configure the malicious logic generator to find and modify the specific variable related to the target variable-frequency drive of a gas centrifuge.

Infection rule 1) Replacing input or output bits with memory bits The infection may make the PLC unresponsive by replacing memory bits with input or output instructions. This approach mainly focuses on the digital values in control logic, which can only result into a static attack with same or similar kind of results when a new rung is appended to the original control logic.

Infection rule 2) Modifying analog control flow determinant We define that a variable X is a control flow determinant if X influences a decision at a conditional branch of control logic. For example, analog control flow determinants include variables of function blocks such as timer preset and counter preset. The modification is done by following steps: 1) finds all analog control flow determinants except input/output variables in control logic, 2) if there is configured range of target variables, finds matching variables among the control flow determinants and modifies them with configured values, 3) if there is no configured value or fails to find the matching variable, randomly

selects an analog control flow determinant and modifies it with a random value.

Infection rule 3) Modifying set-points A set-point is a desired value for a process value of systems like a motor frequency of a gas centrifuge. In most of control logic program, a set-point of underlying physical process is represented using special function blocks or instructions provided by an engineering software. For example, Schneider Electric SoMachine-Basic provides drive block which allow drive devices to be controlled by a PLC. Allen-Bradley's RSLoigx 500 supports proportional integral derivative instructions which control a process variable to be at a desired set-point using a closed process loop. The modification is done by following steps: 1) finds those function blocks or instructions which contain a set-points, 2) if there is configured range of target variables, finds matching variables among the set-points and modifies them with configured values, 3) if there is no configured target value or fails to find the matching variable, randomly selects a set-point and modifies it with a random value.

Infection rule 4) Replacing operators in operational blocks Changing the operational block equation can change the flow of the entire control logic. This modification can be accomplished by replacing the operators with an opposite operator (like := can be replaced with <>) which makes the control flow to act exactly opposite than it was originally intended to.

Injecting Malicious Logic via the Rules We apply the rules to inject three different malicious logic in a control logic program. To generate the *first* malicious logic, the `CLIK` applies the rule-1 to replace an I/O register of an actuator with a memory I/O to control the actuator state via a memory location. In the *second* malicious logic, the `CLIK` applies the rule-5 to append a new rung at the end to energizing an output used in the control logic.

To generate the *third* malicious logic, the `CLIK` applies the rule-4 to disable the condition that energizes an actuator when a certain threshold is met. This logic can cause severe damage to an ICS system. For instance, consider a scenario where a control logic energizes a red light when a particular threshold is reached to indicate danger. If the attacker manipulates the control logic using this method, it will disable the condition to energize the light thereby, making the operator utterly oblivious to the situation.

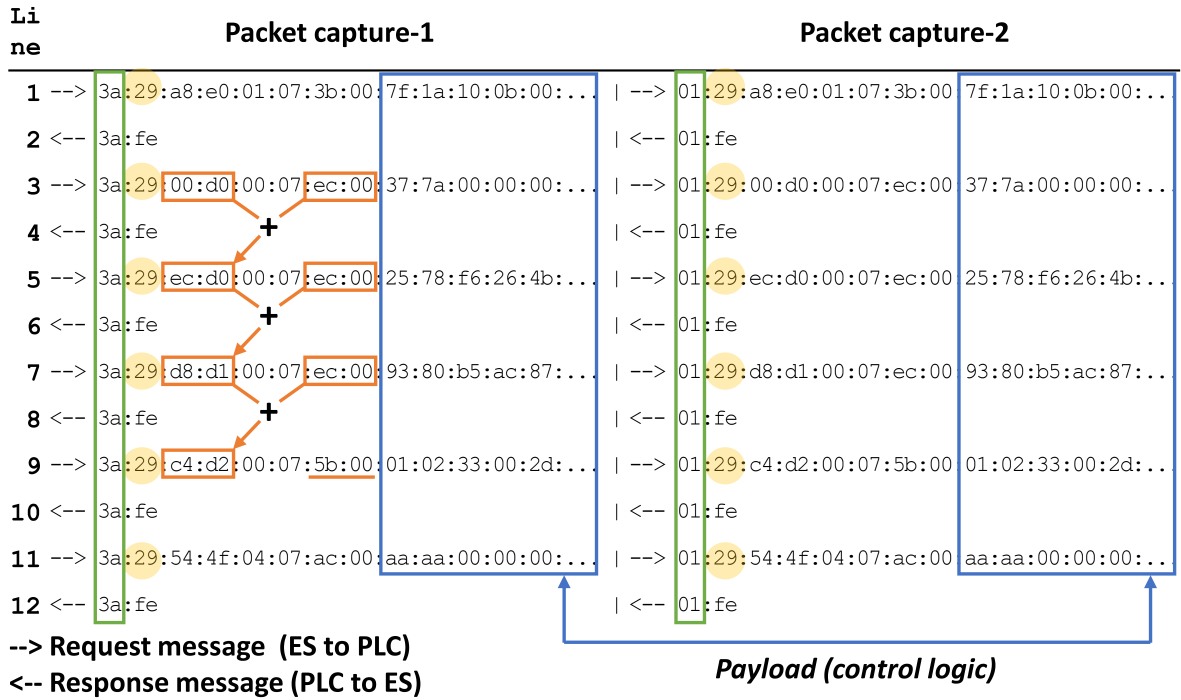
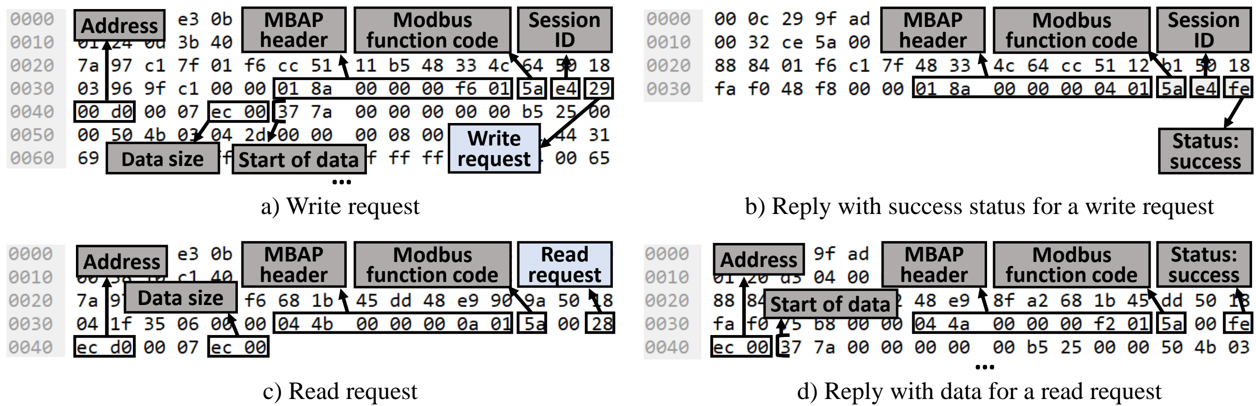


Figure 4.10: A snippet of aligned M221 protocol message chunks

4.6 Virtual M221 PLC

We have developed a virtual M221 PLC based on the systematic approach presented in Section 3.4. Since the Modbus/TCP protocol encapsulates the M221 PLC protocol, the encapsulated PLC protocol messages can be extracted and grouped according to the transaction identifier of the Modbus/TCP protocol. After deduplication of the extracted messages, the remaining message of different packet captures (containing identical control logic) are aligned to facilitate recognition of the *essential protocol format*. Figure 4.10 shows a snippet of aligned message chunks using the `diff` [1] on two packet captures (of same control logic being downloaded to the M221 PLC). The first byte of each message can be inferred as a session related field because its values are identical within a packet capture but different across the packet-captures. The second byte of request messages can be recognized as a write command (remember that the packets are captured when control logic is being downloaded in this example) since its values are always `0x29` in the packet captures. Also, we can find a data-size field by recognizing that the two-byte little-endian value of the seventh and eighth bytes of the request messages exactly represents the size of the following bytes (payload).



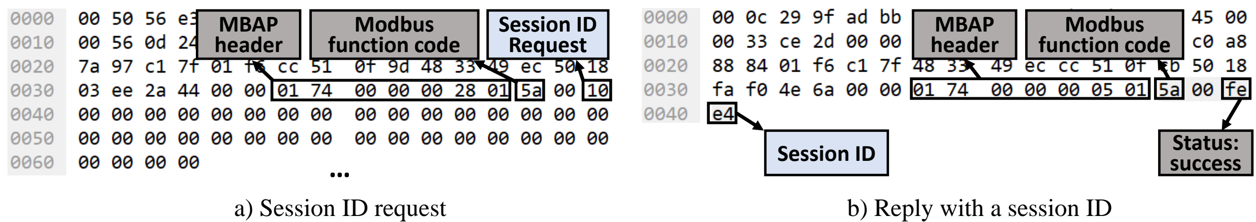
a) Write request

b) Reply with success status for a write request

c) Read request

d) Reply with data for a read request

Figure 4.11: Read and write message format of the M221 proprietary protocol



a) Session ID request

b) Reply with a session ID

Figure 4.12: Session ID request/response message format of the M221 proprietary protocol

The address field (third and fourth bytes in the request messages) can also be easily noticeable since the values are increased by the data size in some successive request messages (line 5,7, and 9 in Figure 4.10). It cannot be recognized as a type of sequence number field because its value is not always increased by the data size of the previous request message. For example, its value (`0x4f54`) of the request message in the line 11 is not the sum of its value (`0xd2c4`) and the data size (`0x5b`) of the previous request message (in line 9).

Figures 4.11, 4.12 and 4.13 show the recognized *essential protocol format* of the M221 PLC. When the virtual PLC receives a read or write request message, it dynamically generates a valid response message using the read/write message formats (Figure 4.11) and the control logic blocks obtained in phase 1 of `CLIK`. Figure 4.12 describes the session ID request and response message format. The session ID is used in the *write* request/response messages and some of the messages in the communication template. Since the M221 PLC decides a session ID for a session, we use the same session ID appeared in the communication template for every session, thereby the virtual PLC does not have to correct the session ID in the communication template.

Figure 4.13 shows the request/response messages used by SoMachine-Basic to check the in-

a) Checksum in *conf1* block

b) Checksum request-1

c) Checksum reply-1

d) Checksum request-2

e) Checksum reply-2

Figure 4.13: Integrity check of control-logic using a checksum in M221 proprietary protocol

tegrity of control logic periodically. We found two types of request/response messages as shown in Figure 4.13. The checksum value is located at the end of the `conf1` block (refer to Figure 4.13(a)). The byte order of the checksum is slightly different for both message types. The checksum reply messages in the *communication template* are corrected according to the checksum in the `conf1` block when the virtual PLC loads the template. Along with the *essential protocol format*, we derive a *communication template* for the M221 PLC according to the procedure explained in Section 3.4. The communication template consists of 40 messages (20 request and response pairs) including two session id related messages and four messages of the integrity checking of control logic. To redirect the packets from the engineering software to the virtual PLC, we use ARP poisoning for the proof-of-concept along with the destination network address translation (DNAT) of `iptables` [3] in the virtual PLC.

Chapter 5

Evaluation

5.1 Experimental Settings

Lab Setup We evaluate CLIK on Schneider Electric Modicon M221 PLC (firmware v.1.6.0.1) and SoMachine-Basic v1.6. The PLC is connected with simple physical processes; each consists of toggle switches, push buttons, pilot lights, potentiometer, ammeter, etc., and communicates with SoMachine-Basic over Ethernet. SoMachine-Basic runs on Windows 7 virtual machine (VM), and the CLIK implementation runs on Ubuntu 16.04 VM.

Dataset Our dataset consists of 52 IL programs for evaluation. The programs have 286 rungs and 1678 instructions in total and are written for different physical processes such as traffic light, gas pipe line and Hot water tank using various conditional and operational instruction blocks.

5.2 Reliability of the Password Attack

We evaluate the password attack using the 52 different control logic programs. Each program is set with a unique password and downloaded to a real M221 PLC. We use a random password generator [4] for each password to generate strong passwords that are 25 characters long including lower case, upper case, and numbers. We performed the attack on the control logic programs and found that the attack always resets the password successfully within 14 seconds. Table 5.1 summarizes the experimental results. The total block size represents the total sum of the six control logic block sizes (`conf1`, `conf2`, `zipHash`, `code`, `data1`, and `data2`).

While executing the attack, the initial target address is set to `0xdfe0` and the `zipHash` block is always mapped to `0xd000`. Therefore, if the size of the `zipHash` block is larger than 4,064 (0xfe0) bytes, the attack would not work. However, according to our experiment result, the size of the `zipHash` block is always smaller than 4,064 bytes. Also, Every `code` block is mapped to a

Table 5.1: The experimental results of the password reset attack

Total block size(KB)	# of project files	Max size zipHash (byates)	Lowest addr. of code block (hex)	Avg. # of write	Max # of write	Avg. time (sec)	Max time (sec)	Attack success rate
6 ~ 7	5	831	0xe088	3325	3413	13.48	13.88	100%
7 ~ 8	19	1712	0xe08c	2943	3266	11.89	13.31	100%
8 ~ 9	25	2261	0xe08c	2034	2385	8.21	9.64	100%
>9	3	3103	0xe26c	1468	2379	5.89	9.42	100%
Total	52	3103	0xe088	2458	3413	9.93	13.88	100%

random address after `0xe000`, thereby we confirm that the overwriting the target memory locations of the PLC during the attack does not affect the integrity of the `code` block. Finally, we evaluate the attack on two different firmware versions of the PLC i.e., 1.5.1.0 and 1.6.0.1. The attack is successful on both versions. Thus, we confirm that both firmware versions are vulnerable to this attack.

5.3 Compilation & Decompilation Accuracy

We evaluate the accuracy of the decompiler `Eupheus` and its counterpart compiler using the dataset of 52 IL programs. We download programs to a real M221 PLC, capture their network traffic and then, extract them from the traffic. The control logic programs are RX630 machine code that run the PLC. We use `Eupheus`to decompile the programs into their IL source code and then, compare the decompiled and original IL code to measure the accuracy of `Eupheus`. The experimental results are summarized in Table 5.2 and conclude that `Eupheus` can accurately decompile RX630 machine code into IL code since it does not encounter any intermingling of data and code.

To measure the accuracy of the compiler, we compiled the decompiled version back to RX630 machine code and compare both (recompiled and original) versions. The experimental results show that the compiler can accurately recompile the decompiled version into RX630 machine code.

5.4 Generation of Malicious Logic

We evaluate the generation of malicious logic using three rules on the IL programs (traffic light, gas pipeline, hot water tank, and others) in the datasets. Table 5.4 includes the experimental results

Table 5.2: The accuracy of the decompiler `Eupheus`

Symbol (Name)	SoMachine Basic	Eupheus	Accuracy	Symbol (Name)	SoMachine Basic	Eupheus	Accuracy
LD (Load value)	339	339	100%	DIV (Division)	11	11	100%
LDN (Load negated)	84	84	100%	OR (Bitwise OR)	36	36	100%
OTE (Output energize)	251	251	100%	AND (Bitwise AND)	136	136	100%
MUL (Multiplication)	31	31	100%	M (Memory bit)	197	197	100%
TON (Timer on delay)	24	24	100%	S (Set)	30	30	100%
TOF (Timer off delay)	4	4	100%	OUT_BLK (Block out)	44	44	100%
CTU (Count up)	8	8	100%	END_BLK (End of the block)	44	44	100%
R (Reset)	21	21	100%	IN (Input-timer)	27	27	100%
END (End of control logic)	52	52	100%	LDR (Load rising edge)	17	17	100%
CTD (Count down)	8	8	100%	LDF (Load falling edge)	16	16	100%
EQU (Equal)	13	13	100%	DR (Drum register)	8	8	100%
GEQ (Greater than or equal)	4	4	100%	MW (Memory word)	71	71	100%
GRT (Greater than)	9	9	100%	TP (Pulse timer)	8	8	100%
LEQ (Less than or equal)	5	5	100%	MF (Memory float)	76	76	100%
LES (Less than)	3	3	100%	Short (Short)	10	10	100%
NEQ (Not equal)	4	4	100%	SB (System bit)	26	26	100%
ADD (Addition)	16	16	100%	XOR (Bitwise XOR)	2	2	100%
SUB (Subtraction)	8	8	100%	Done (Done)	1	1	100%
NOTE (Negated output energize)	33	33	100%	Write_Var (Write Data to a Modbus Device)	1	1	100%
Total	1678	1678	100%				

of infecting the programs successfully.

The first infection is performed on Traffic light program that changes the I/O instructions to energize the output. We trace the I/O instructions in the decompiled code and the location of every I/O instruction is noted. The goal of the infection is to illuminate all LEDs. Thus, the malicious Logic generator redirects every input instruction to negated memory bit input. Since the default Memory bit value is `false`, the negated memory bit input will result in true thereby illuminating all lights used in the Traffic light signal.

Table 5.3: The experiment results on the virtual M221 PLC

Total block size(KB)	# of control logic	Avg. # of template lookup	Avg. # of dynamic generation	Avg. time (sec)	Upload success rate
6 ∼ 7	5	105	35	10.82	100%
7 ∼ 8	19	110	38	12.12	100%
8 ∼ 9	25	109	41	12.09	100%
>9	3	115	54	12.15	100%
Total	52	109	40	11.98	100%

The second infection inserts a new rung into the IL program of hot water tank. The program controls the inlet and outlet valves, can fill or empty the tank, and maintains the water temperature between the values. After getting the decompiled IL code, we infected it by appending a rung, which has one output LED (L1) as an input instruction and one random LED (L2) as output instruction. The LEDs indicate different hazard situations such as too hot or full tank. After appending the rung, when the L1 is energized as per the program, L2 will also be energized, which causes a confusion as to which situation is true.

The third infection modifies the gas pipeline program, which maintains the analog flow determinants. This attack finds the determinants and changes the operation block by modifying the decision operators such as (< or >) to the assignment operator (=). The modification causes an unexpected behavior in the operation of the PLC.

5.5 Effectiveness of the M221 Virtual PLC

To evaluate the virtual M221 PLC, we first download each control logic program to the virtual PLC, and extract control logic blocks from it. Then, we verify whether the virtual PLC can successfully upload the control logic blocks to SoMachine-Basic, thereby SoMachine-Basic decompiles it and show its source code. Table 5.3 shows the experimental results. The number of dynamic generation means the number of times read/write response messages are generated dynamically (note that control logic read/write messages are not in the communication template) and the time represents the total operation time of the virtual PLC to successfully upload control logic to SoMachine-Basic. In the experiments, virtual PLC uploads every control logic successfully to SoMachine-Basic.

Table 5.4: The final evaluation result on CLIK

Control logic	Real world logic	# of files	Original logic size(bytes)	Infected logic size(bytes)	Phase-I Stealing	Phase-II Decompile	Phase-III Infection	Phase-IV Concealing	CLIK Success rate
Traffic Light	Yes	1	8407	8415	100%	100%	100%	100%	100%
Gas Pipeline	Yes	1	10110	10110	100%	100%	100%	100%	100%
Hot Water Tank	Yes	1	7241	7245	100%	100%	100%	100%	100%
Others	No	49	(avg.) 8029	(avg.) 8034	100%	100%	100%	100%	100%
Total	-	52	(avg.) 8061	(avg.) 8070	100%	100%	100%	100%	100%

5.6 Putting CLIK-Phases All Together

We evaluate the entire CLIK attack by running it in autonomous mode. Table 5.4 summarizes the evaluation result. We reuse the 52 control logic program in the dataset for the evaluation. Each control logic is protected by a unique random password. After a complete execution of CLIK we reset the environment before initiating the CLIK attack for the next control logic. The evaluation results show that CLIK conducts its four phases (stealing, decompiling, infection, concealing) for each control logic successfully, as shown in Table 5.4.

Chapter 6

Conclusion

We presented `CLIK`, a new type of autonomous attack on the control logic of a PLC in industrial control systems to disrupt a physical process controlled by the PLC. `CLIK` as a full attack was implemented and evaluated on a real PLC (Schneider Electric Modicon M221) and engineering software (SoMachine-Basic). The implementation of the `CLIK` included exploiting a zero-day vulnerability to subvert a password authentication used in the PLC, a decompiler `Eupheus` to transform binary control logic into its corresponding IL source code, and a virtual PLC to engage the SoMachine-Basic using the acquired network traffic of normal control logic. `CLIK` utilized a rule-based approach for automated malicious modification of the source code. The evaluation results of `CLIK` on 52 control logic program showed that it automatically infected control logic of the M221 PLC and hid the infection from SoMachine-Basic with 100% success rate. This thesis work has been submitted to ACSAC 2018 conference.

Bibliography

[1] GNU Diffutils. https://www.gnu.org/software/diffutils/, 2017.

[2] Ettercap. https://ettercap.github.io/ettercap/, 2018.

[3] Netfilter. https://netfilter.org/, 2018. [Online; accessed 28-Mar-2018].

[4] Strong Random Password Generator. https://passwordsgenerator.net/, 2018.

[5] Ali Abbasi and Majid Hashemi. Ghost in the plc designing an undetectable programmable logic controller rootkit via pin control attack. *Black Hat Europe*, pages 1–35, 2016.

[6] I. Ahmed, S. Obermeier, M. Naedele, and G. G. Richard III. SCADA Systems: Challenges for Forensic Investigators. *Computer*, 45(12):44–51, Dec 2012.

[7] I. Ahmed, S. Obermeier, S. Sudhakaran, and V. Roussev. Programmable Logic Controller Forensics. *IEEE Security Privacy*, 15(6):18–24, November 2017.

[8] I Ahmed, V Roussev, W Johnson, S Senthivel, and S Sudhakaran. A SCADA System Testbed for Cybersecurity and Forensic Research and Pedagogy. In *Proceedings of the 2nd Annual Industrial Control System Security Workshop*, ICSS '16, pages 1–9, New York, NY, USA, 2016. ACM.

[9] Nicolas Falliere, Liam O Murchu, , and Eric Chien. W32.stuxnet dossier. Technical report, 2011.

[10] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *24th Network and Distributed System Security Symposium*, 2017.

[11] Stephen McLaughlin and Patrick McDaniel. Sabot: specification-based payload generation for programmable logic controllers. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 439–449. ACM, 2012.

[12] Stephen E McLaughlin. On dynamic malware payloads aimed at programmable logic controllers. In *HotSec*, 2011.

[13] Olivera Pavlovic, Ralf Pinger, and Maik Kollmann. Automated formal verification of plc programs written in il. In *Conference on Automated Deduction*, pages 152–163, 2007.

[14] Renesas Electronics. *RX Family User's Manual:Software*, 2013.

[15] S Senthivel, I Ahmed, and V Roussev. SCADA network forensics of the PCCC protocol. *Digital Investigation*, 22:S57–S65, 2017.

[16] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. Denial of Engineering Operations Attacks in Industrial Control Systems. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 319–329, New York, NY, USA, 2018. ACM.

Vita

Sushma Kalle received her Bachelor Degree in Technology from N.B.K.R.I.S.T, India in 2012. She joined the University of New Orleans for Computer Science Master of Science program in Fall 2016. She started working as a Research Assistant in Cy-Phy Laboratory at University of New Orleans under the supervision of Dr. Irfan Ahmed. She have been doing active research in Cyber Physical Systems and analysing on the Network protocols used by these systems. Her research interest includes Cyber security, Network Forensics, Memory Analysis, and Reverse Engineering.