CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC

Sushma Kalle
University of New Orleans
skalle1@uno.edu

Nehal Ameen
University of New Orleans
nameen@uno.edu

Hyunguk Yoo
University of New Orleans
hyoo1@uno.edu

Irfan Ahmed*
Virginia Commonwealth University
iahmed3@vcu.edu

***Abstract*—This paper presents CLIK, a new remote attack on the control logic of a programmable logic controller (PLC) in industrial control systems. The control logic defines how a PLC controls a physical process such as a nuclear plant. A full control logic attack faces two critical challenges: 1) infecting the control logic in a PLC at a field site and, 2) hiding the infection from engineering software at a control center since the software can obtain the infected logic from the PLC and reveal it to a control engineer. The existing academic efforts only (partially) address the former. CLIK is a first practical control-logic attack that deals with both challenges successfully. It modifies the control logic running in a remote target PLC automatically to disrupt a physical process. CLIK also employs a new virtual PLC approach that hides the malicious modifications by engaging the engineering software with a captured network traffic of the original (uninfected) control logic. It is fully implemented on real hardware/software used in industrial settings and is made publicly available for academic research on control logic attacks[1]. CLIK consists of four phases and takes less than a minute to complete an attack cycle. As part of the implementation, we found a critical (zero-day) vulnerability in the password authentication mechanism of a target PLC, which allows the attacker to overwrite password hash in the PLC during the authentication process and gain access to the (protected) control logic. We have disclosed the vulnerability responsibly to the PLC vendor who has already patched the vulnerability[2].**

I. 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 [16], [17], [21], [23], [38], [24], [29], [5], [7], [42]. To secure ICS environments, it is imperative to understand their threat vectors, i.e., how an adversary can target these systems.

An ICS environment consists of a control center and field sites. 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 [18]. 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 [24] 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 full control logic attack faces two critical challenges: First is the malicious modifications of the control logic in a target PLC at a field site. Second is the hiding of the infected logic from the engineering software at the control center, which can acquire the logic from the PLC remotely and reveal the infection to a control engineer. The existing academic efforts on control logic attacks such as SABOT [33], Ladder Logic Bomb [26], Dynamic Payloads [32], PLCinject [31], and PLC-Blaster [41] (partially) focus on the former challenge and do not consider the latter. Furthermore, the current real-world attacks heavily rely on exploiting engineering software to infect the control-logic, which is a significant limitation and requires access to the software in a target environment before launching the attack. For instance, Stuxnet compromises `STEP 7` engineering software (by replacing `s7otbxdx.dll` that handles communication with the PLC) to infect the logic.

Attack Scenario. In this paper, we present CLIK, an autonomous full attack-chain on the control logic of a PLC. We assume a realistic attack 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 . 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

*Ahmed completed this work while he was at the University of New Orleans

[1]https://gitlab.com/hyunguk/clik

[2]https://ics-cert.us-cert.gov/advisories/ICSA-18-240-01

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23xxx
www.ndss-symposium.org

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 USB device, vulnerable web server, etc.

Proposed Attack. CLIK consists of four phases; It involves compromising the PLC security measures and stealing the control logic from a 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. The final phase hides the malicious logic in the PLC from the engineering software. CLIK employs a new virtual-PLC approach utilizing a 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. Each CLIK phase is an individual attack i.e., data exfiltration (phase 1), reconnaissance (phase 2), infection (phase 3), and stealth (phase 4). CLIK combines these attacks systematically to create a complete attack-chain.

We implement CLIK on real hardware/software used in industrial settings and make it publicly available to facilitate academic research [13]. As part of implementation, CLIK exploits a critical (zero-day) vulnerability in the password authentication mechanism of a target PLC and contains a decompilation capability referred to as Eupheus for the Instruction List (IL) defined by IEC 61131-3 [2] to program PLCs.

We evaluate CLIK on 52 control logic programs of real-world physical processes such as traffic light, gas pipeline, and hot water tank. The evaluation results show that CLIK completes its attack cycle in less than a minutes successfully.

Contributions. We summarize the paper contribution as:

- *Full attack-chain.* We design a new complete attack-chain for infecting control logic including vulnerability exploitation, decompilation, malicious logic generation, and concealment of infection via a novel virtual-PLC approach.
- *Practical implementation.* We implement a first full practical attack on control logic to facilitate academic research on this topic. The attack is performed successfully on real ICS hardware/software used in industrial settings.
- *Decompiler.* We develop a decompiler that transforms a low-level control-logic in RX630 microcontroller instructions [37] to a high-level instruction list (IL) program. Although the decompiler is used for the attack, it can be applied to a variety of security applications such as PLC Code Analytics [43], [34], and ICS network forensics [39], [40].
- *Critical zero-day vulnerability.* We present a successful exploitation of a critical (zero-day) vulnerability on password authentication mechanism of Modicon M221 PLC. M221 is a compact controller introduced by Schneider Electric in August 2014 to replace its Twido controllers [4]. The PLC represents the latest technology to meet the requirements of the Industry 4.0 trend of automation and data exchange in manufacturing.

II. Control Logic Infection AttacK (CLIK)

Figure 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.

A. Phase I: Stealing the Control Logic from a Target PLC

The *first* phase involves gaining access to a target PLC and retrieves the control 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 white-listing of IP addresses, and *authentication* (such as password) to restrict remote read/write access to PLC.

In Section III-A, 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 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 obtains the configuration blocks to get valid mapping addresses and sizes of other blocks.

B. 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 2 illustrates the decompilation process of CLIK, which takes the code block as an input and utilizes a database of binary code to instruction (defined in IEC 61131-3) 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.

C. Phase III: Infecting 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.

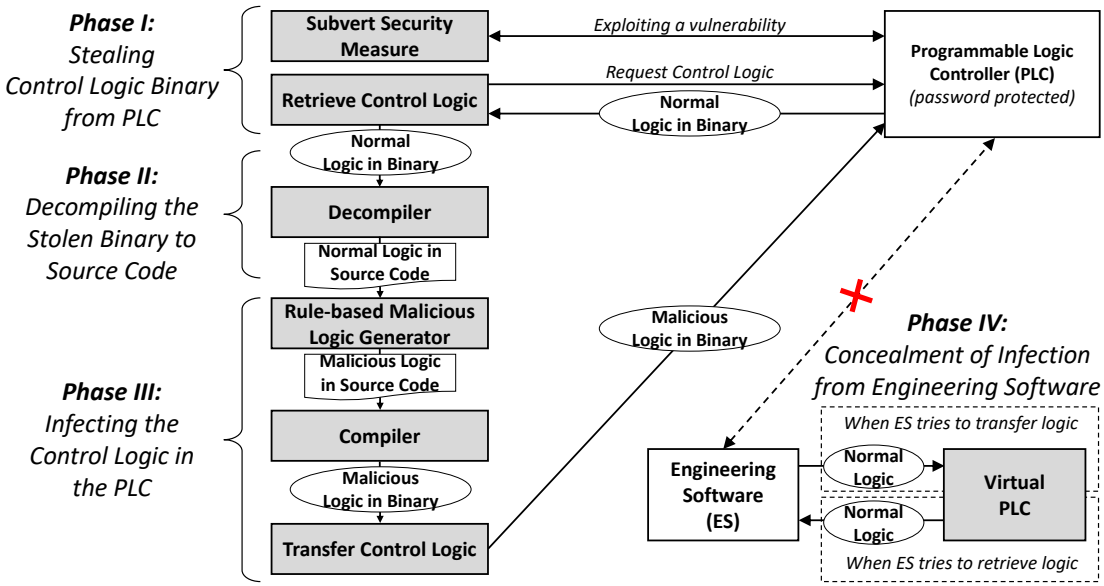


Fig. 1: High-level overview of CLIK, a control logic infection attack

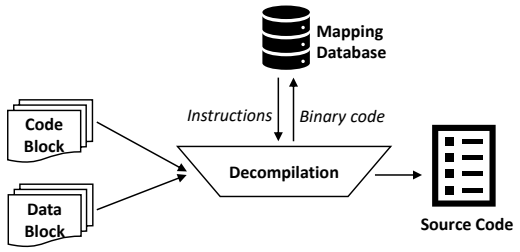


Fig. 2: Decompilation of the Control Logic

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. 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 specific 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 (which are variables to influence a decision at a conditional branch of control logic), insert/delete instructions/rungs, etc. For example, similar to Stuxnet (but without a pregeneration of malicious payload), 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.

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 binary code for a high level instruction in the database and replaces the latter with the former.

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 blocks of the infected logic should be mapped to the address of the original blocks to ensure the stable transition of the PLC to the infected logic.

D. 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.

Virtual PLC. We present a new infection hiding method based on a virtual PLC. 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 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. Figure 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 *dynamic field format* in two stages for a target PLC. It executes both stages offline during the preparation of the CLIK attack.

We build virtual PLC based on our insight that the regular traffic (excluding messages containing control logic in their

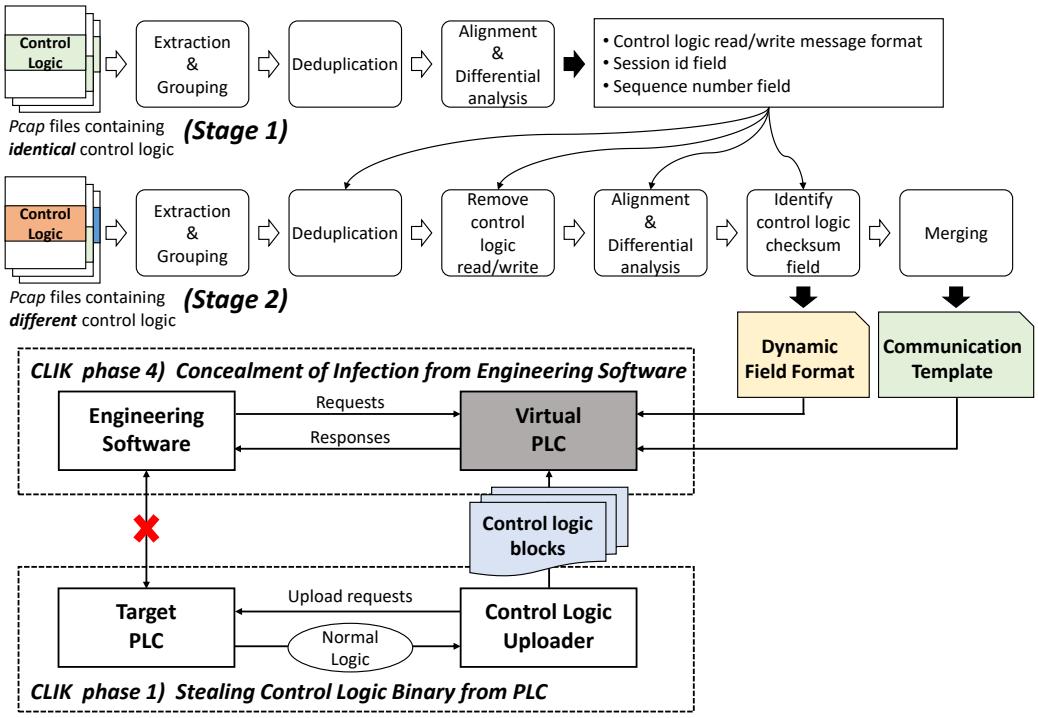


Fig. 3: A systematic approach to build a virtual PLC

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

Deriving Dynamic Field Format. Stage 1 collects multiple network packet captures of the same control logic and then, processes each packet capture as follows: encapsulated PLC 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 [1]. 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 messages since the messages contain the same control logic, which leads to find some dynamic fields of a proprietary protocol. This includes 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/response messages involving control logic read/write are removed based on the dynamic field 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) such as checksum fields. These fields, as well as the previously derived format in stage 1, compose *dynamic field format*. Lastly, in the merging step, the messages are merged ignoring all the derived dynamic fields to make a communication template consisting of the unique request and response pairs.

During the attack, a virtual PLC finds the corresponding messages to a request message using the *communication template* and then, corrects the dynamic fields using the *dynamic field 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`.

III. 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.

A. 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. We found a zero-day vulnerability in the authentication mechanism and further developed a proof-of-concept exploit to subvert the password protection. We confirmed that the latest versions of firmware (v1.5.1 and v1.6.0) and SoMachine-Basic (v1.5 and v1.6) are impacted by the vulnerability. We disclosed the vulnerability responsibly to the PLC vendor who already patched it. ICS-CERT issued the advisory and three CVEs [8].

Figure 4(a) 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 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 is to identify the correct location of the hash code in the PLC.

Figure 4(c) 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 [37], 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`.

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.

Algorithm 1 Pseudocode for password reset attack

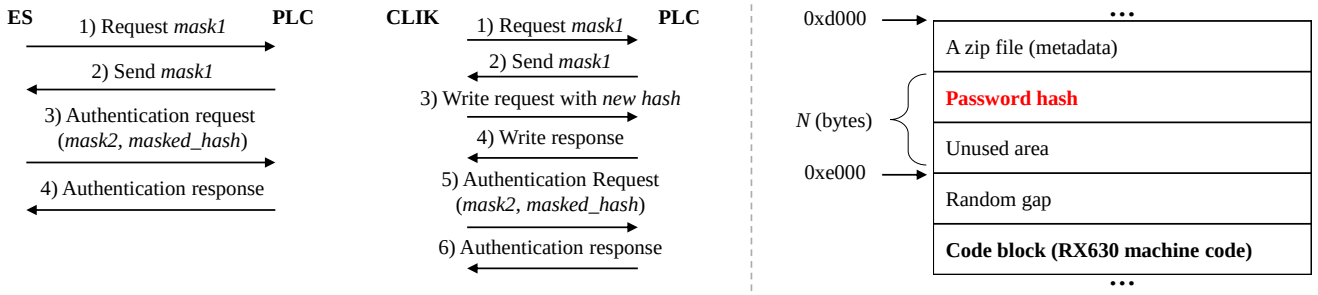
```
Input: New password
Result: Reset password with the new password
1:  newHash ← sha-256 hash of the new password
2:  mask1 ← Request mask1 from the PLC
3:  mask2 ← A random number between 0 and 255
4:  targetAddr ← 0xdfe0
5:  hashSize ← 32
6:  for i = 0 to hashSize − 1 do
7:      maskedHash[i] ← newHash[i]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

Figure 4(b) 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 and then, sends an authentication request to the target PLC using *mask2* and *maskedHash* (line 12 in Algorithm 1). 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.

B. Control Logic Uploader & Downloader

The *Uploader* and *Downloader* retrieves the control logic from and transfer it to a PLC respectively. In the 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. 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 requests to the PLC to retrieve the entire logic. Figure 5 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.



a) Normal password authentication

b) Password authentication while exploit the vulnerability

c) The location of the password hash in the address space of the M221 PLC

Fig. 4: Exploiting password authentication between Modicon M221 (PLC) and SoMachine Basic Engineering Software (ES)

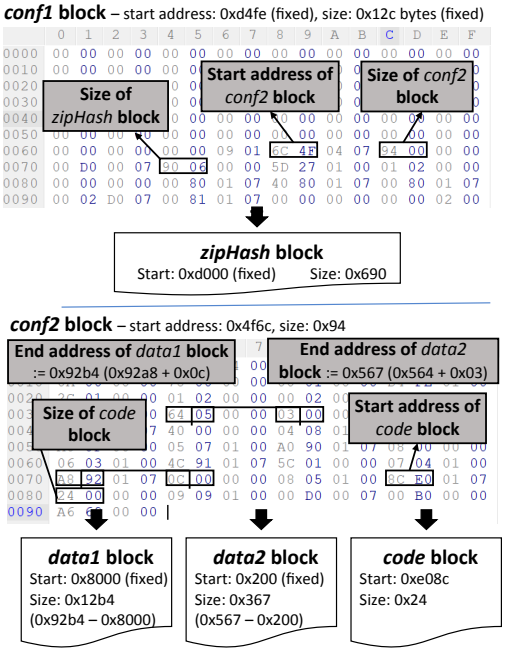


Fig. 5: Start address and size of M221 control logic blocks

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.

C. 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 machine 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 6). Currently, the database consists of 4079 mappings for 21 types of different instructions including input and output, relative branch, function 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. 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.

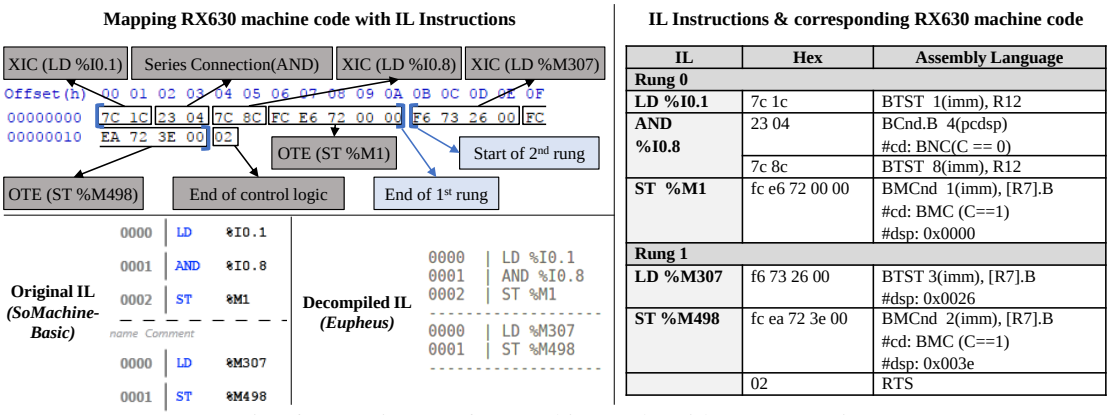
The Figure 6 shows an example of RX630 control logic that is mapped to IL instructions. The rungs are separated when an output instruction is preceded by the instruction of an input or a function block such as `Timer` or `Counter`. In the latter case, the rung ends with `END_BLK` instruction denoted as `0x7F1A11` in RX630 machine code. After the rungs are separated, `Eupheus` decompiles each rung at a time by identifying the instructions in the database and replacing the RX630 machine code with equivalent IL instruction. The logical operators `AND` and `OR` are denoted in binary control logic as `0x23` and `0x22` respectively, followed by the byte that indicates the total number of bytes of the next instruction. For example, `0x7C0C23047C1C` represents that the two `LD` instructions `0x7C0C` and `0x7C1C` are in series connection (`AND`) and the `0x04` shows that the next instruction has two bytes.

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

D. Malicious Logic Generator

`CLIK` employs a rule-based approach to inject malicious logic in the decompiled IL code automatically. The decompilation ensures that `CLIK` understands the binary control logic (RX630 machine code) correctly since the opcode of each instruction has different size; a longer opcode may contain one or more smaller opcodes. Furthermore, the decompiled IL code exposes the structure and semantics of control logic. Thus, it is easier to parse and manipulate the control logic for an IL code than RX630 machine code.

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



IL Instructions & corresponding RX630 machine code

IL	Hex	Assembly Language
Rung 0		
LD %I0.1	7c 1c	BTST 1(imm), R12
AND %I0.8	23 04	BCnd.B 4(pcdsp) #cd: BNC(C == 0)
	7c 8c	BTST 8(imm), R12
ST %M1	fc e6 72 00 00	BMCnd 1(imm), [R7].B #cd: BMC (C==1) #dsp: 0x0000
Rung 1		
LD %M307	f6 73 26 00	BTST 3(imm), [R7].B #dsp: 0x0026
ST %M498	fc ea 72 3e 00	BMCnd 2(imm), [R7].B #cd: BMC (C==1) #dsp: 0x003e
	02	RTS

Fig. 6: Mapping RX630 machine code with IL Instructions

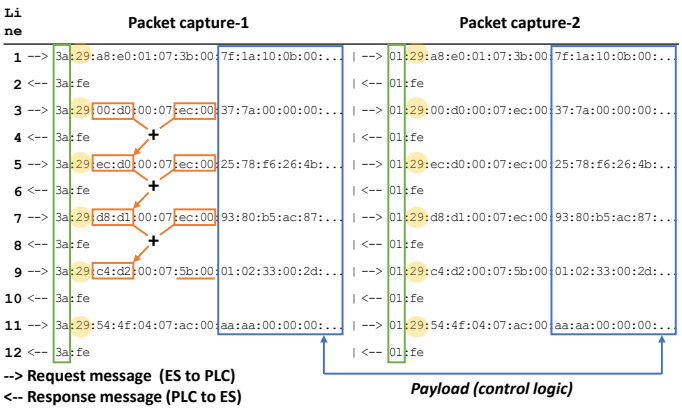


Fig. 7: Alignment of M221 protocol message chunks

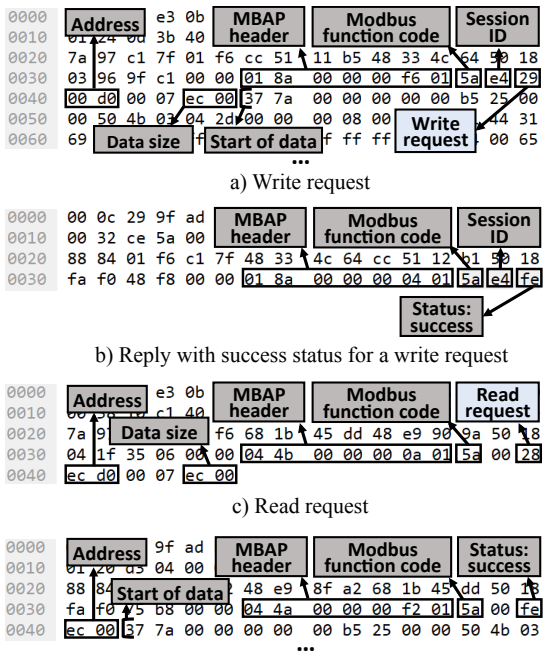
them with memory bits. *Rule-2:* If an analog variable which affects control flow of the logic then, modify the analog value. *Rule-3:* If a set-point is in a pre-configured (estimated) range then, modify the set-point value. *Rule-4:* If an operational block has an equation then, replace an operator in the equation. For instance, the operator :=(equal) can be replaced with <>(not equal). *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.

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.

E. Virtual M221 PLC

We have developed a virtual M221 PLC based on the systematic approach presented in Section II-D. Since the



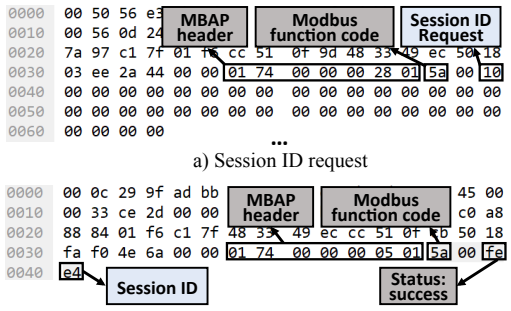
a) Write request

b) Reply with success status for a write request

c) Read request

d) Reply with data for a read request

Fig. 8: Read/write message format of the M221 protocol



a) Session ID request

b) Reply with a session ID

Fig. 9: Session ID request/response message format of the M221 proprietary protocol

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 *dynamic field format*. Figure 7 shows a snippet of aligned message chunks using the `diff` utility [10] 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 id 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). 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 7). 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.

Figures 8, 9 and 10 show the recognized *dynamic field 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 8 and the control logic blocks obtained in phase 1 of `CLIK`. Figure 9 describes the session id request and response message format.

Figure 10 shows the request/response messages used to check the integrity of control logic periodically. We found two types of request/response messages as shown in Figure 10. The checksum value is also located at the end of the `conf1` block (refer to Figure 10(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 *dynamic field format*, we derive a *communication template* for the M221 PLC according to the procedure explained in Section II-D. 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` [11] in the virtual PLC.

IV. Evaluation

A. 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` runs on a Ubuntu 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

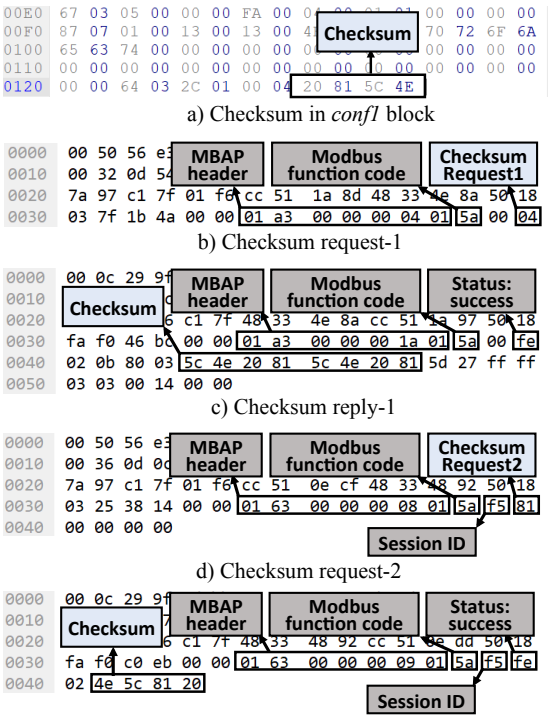


Fig. 10: Integrity check of control-logic using a checksum in M221 proprietary protocol

traffic light, gas pipe line and Hot water tank using various conditional and operational instruction blocks.

B. 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 [12] 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 IV (in Appendix) summarizes the experimental results. The total block size represents the total sum of the six control logic block sizes (`conf1`, `conf2`, `zipHash`, `code`, `data1`, `data2`).

While executing the attack, the initial target address is set to `0xdfe0` and the `zipHash` block is always mapped to `0xd000`. Every `code` block is mapped to a 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.

C. 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

TABLE I: 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%				

experimental results are summarized in Table I 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 compile 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.

D. 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 III includes the experimental results 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.

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 include 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, causing a confusion between the two situations.

TABLE II: 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
$6 \sim 7$	5	105	35	10.82	100%
$7 \sim 8$	19	110	38	12.12	100%
$8 \sim 9$	25	109	41	12.09	100%
>9	3	115	54	12.15	100%
Total	52	109	40	11.98	100%

The third infection modifies the gas pipeline program, which contains 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 PLC operations.

E. 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 II 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, the virtual PLC uploads every control logic successfully to SoMachine-Basic.

TABLE III: The complete CLIK execution of all four phases

Control logic	# of files	Original logic size(bytes)	Infected logic size(bytes)	CLIK Success rate
Traffic Light	1	8407	8415	100%
Gas Pipeline	1	10110	10110	100%
Hot Water Tank	1	7241	7245	100%
Others	49	8029 (avg.)	8034 (avg.)	100%
Total	52	8061 (avg.)	8070 (avg.)	100%

F. Putting CLIK-Phases All Together

We evaluate the entire CLIK attack by running it in autonomous mode. Table III 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 III. On average, CLIK takes less than a minute to complete an attack cycle and infect a target control logic.

V. Countermeasures

We suggest countermeasures to CLIK at three levels: protection, verification, and detection of control logic.

Control logic protection. The effective approaches to prevent unauthorized access and modifications to control logic are by improving the isolation of ICS from other networks [30], compliance with standard security practices [28], [9], and defense-in-depth security in control systems [6]. Furthermore, PLC vendors secure their firmware by digital signature or restricting firmware updates to only USB interface. They can employ similar approach to control logic. Unfortunately, the security measures in PLCs largely rely on their obscurity, which should be replaced with strong security solutions for authentication, access control, key management, etc.

Control logic verification. A plausible solution is to verify control-logic before downloading it to a PLC. For instance, comparing the control logic being transferred with the normal control logic [39] or applying a model checking technique [43], [20], [36], [35], [19]. In other words, the control-flow of control logic running in a PLC can be checked if the execution of control logic including PLC runtime follows only valid execution paths [15].

Control logic detection. Perhaps, the most effective approach is to detect and block the transfer of control logic packets in an ICS network unless the transfer is explicitly granted. Typically, it can be achieved by checking the PLC protocol header, which contains information about the type of payload [30]. Otherwise, packet payload can be scrutinized to identify control logic or unauthorized changes in PLC configuration [27].

VI. Related Work

This section presents PLC attacks closely related to CLIK.

McLaughlin [32] proposes a malware model that discovers plant devices through the fieldbus protocols [3], infers the properties of the target physical process (e.g., safety interlocking and plant structure) and generates malicious control-logic. McLaughlin and McDaniel [33] further present SABOT as a proof-of-concept tool in the same direction. SABOT infers the configuration of the mapping between PLC's input/output variables and connected physical devices. It uses a process specification technique to describe the behavior of a target physical process and a model checking technique [22] to find the PLC variables to device mapping. These approaches are limited to malicious logic generation and are complementary to CLIK. They can be integrated in the phase 3 of CLIK rule-based malicious logic generation for improved performance. Furthermore, unlike CLIK these approaches are never evaluated on real hardware/software used in industrial settings.

Stuxnet [24], one of the most notorious malware in history, represents a best case for a real-world control logic attack on PLCs. Stuxnet contains four zero-day vulnerabilities of MS Windows and two stolen digital certificates. It infects control logic in target PLCs (Siemens S7-300) by compromising engineering software (Siemens SIMATIC STEP 7). Furthermore, the malicious control logic was precompiled in Stuxnet payload. It is widely assumed that the authors of Stuxnet have been sponsored by nation-states and was cooperated by insiders to get the knowledge of target PLC configurations and semantics of underlying physical processes to facilitate the generation of malicious payload. On the other hand, CLIK does not rely on precompiled malicious payload and employs rule-based approach to make malicious modifications in a benign control logic.

Senthivel *et al.* [40] present three control logic attack scenarios referred to as denial of engineering operations (DEO) attack where an attacker interferes with the normal operation of downloading/uploading of PLC control logic leading to compromise the situational awareness of an operator in control center. In DEO I, the attacker deceives the engineering software when it tries to retrieve the control logic from a PLC. It intercepts the network traffic and removing the infection from the control logic. DEO II is similar to DEO I except that it modifies a legitimate control instruction with junk data such as 0xFFFF, which can crash engineering software. DEO III does not require intercepting the traffic and mostly installs a malformed specially-crafted control logic in a PLC. When the engineering software attempts to acquire the control logic from the PLC, it cannot process the logic causing it to crash. The CLIK on contrary ensures continuity of engineering operations via virtual PLC and does not crash engineering software.

PLC rootkits [14], [25] attack a PLC by compromising its firmware or runtime rather than infecting a control logic. Since they reside below the control logic of a PLC, engineering software is not aware of them. However, unlike CLIK, the rootkits have several limitations. They require a root access to a target PLC, remote code execution vulnerability [14], (malicious) firmware update capability, or physical access to the hardware interfaces in PLCs such as JTAG [25].

VII. 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-chain 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 CLIK can work on different logic programs successfully.

REFERENCES

[1] "MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b," Modbus Organization, Specification, 2006.

[2] "IEC 61131-3 Ed. 3.0 b:2013, Programmable controllers:Programming languages," International Electrotechnical Commission, Standard, 2013.

[3] "IEC 61158-1:2014 Industrial communication networks - Fieldbus specifications: Overview and guidance for the IEC 61158 and IEC 61784 series," International Electrotechnical Commission, Standard, 2014.

[4] "Twido controller phase out - Schneider Electric recommends Modicon M221," https://www.rs-online.com/designspark/twido-controller-phase-out-schneider-electric-recommends-modicon-m221, 2015, [Online; accessed 03-June-2018].

[5] "Cyber-attack against ukrainian critical infrastructure," https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01, 2016.

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

[7] "CRASHOVERRIDE Malware," https://ics-cert.us-cert.gov/alerts/ICS-ALERT-17-206-01, 2017.

[8] "Advisory (ICSA-18-240-01)-Modicon M221," https://ics-cert.us-cert.gov/advisories/ICSA-18-240-01, 2018, online accessed Feb-2019.

[9] "Framework for improving critical infrastructure cybersecurity version 1.1," National Institute of Standards and Technology, Tech. Rep., 2018.

[10] "GNU Diffutils," https://www.gnu.org/software/diffutils/, 2018, [Online; accessed 28-Mar-2018].

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

[12] "Strong Random Password Generator," https://passwordsgenerator.net/, 2018, [Online; accessed 28-Mar-2018].

[13] "CLIK gitlab repo," https://gitlab.com/hyunguk/clik , 2019, [Online; accessed Feb-2019].

[14] A. Abbasi and M. Hashemi, "Ghost in the plc designing an undetectable programmable logic controller rootkit via pin control attack," *Black Hat Europe*, pp. 1–35, 2016.

[15] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "Ecfi: Asynchronous control flow integrity for programmable logic controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC, 2017.

[16] I. Ahmed, S. Obermeier, M. Naedele, and G. G. R. III, "SCADA systems: Challenges for forensic investigators," *Computer*, vol. 45, no. 12, pp. 44–51, Dec 2012.

[17] I. Ahmed, S. Obermeier, S. Sudhakaran, and V. Roussev, "Programmable logic controller forensics," *IEEE Security Privacy*, vol. 15, no. 6, pp. 18–24, November 2017.

[18] 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*, ser. ICSS, 2016, pp. 1–9.

[19] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade. plc: A verification platform for programmable logic controllers," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2012, pp. 338–341.

[20] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of plc programs written in instruction list," in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4. IEEE, 2000, pp. 2449–2454.

[21] A. A. Cárdenas, S. Amin, and S. Sastry, "Research challenges for the security of control systems," in *Proceedings of the 3rd Conference on Hot Topics in Security*, ser. HotSec, 2008, pp. 6:1–6:6.

[22] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[23] D. Dzung, M. Naedele, T. P. Von Hoff, and M. Crevatin, "Security for industrial communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1152–1177, 2005.

[24] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.

[25] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking plcs with physical model aware rootkit," in *24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[26] N. Govil, A. Agrawal, and N. O. Tippenhauer, "On ladder logic bombs in industrial control systems," in *Computer Security*, S. K. Katsikas, F. Cuppens, N. Cuppens, C. Lambrinoudakis, C. Kalloniatis, J. Mylopoulos, A. Antón, and S. Gritzalis, Eds. Cham: Springer International Publishing, 2018, pp. 110–126.

[27] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, "Through the eye of the plc: semantic security monitoring for industrial processes," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2014, pp. 126–135.

[28] R. Halbgewachs, "Control systems security standards," Sandia National Laboratories, Tech. Rep., 2009.

[29] ICS-CERT, "ICS Focused Malware," https://ics-cert.us-cert.gov/advisories/ICSA-14-178-01, 2014.

[30] M. A. A. H. Keith Stouffer, Victoria Pillitteri, "Guide to industrial control systems (ics) security," *NIST special publication*, 2015.

[31] J. Klick, S. Lau, D. Marzin, J.-O. Malchow, and V. Roth, "Internet-facing plcs-a new back orifice," *Blackhat USA*, 2015.

[32] S. McLaughlin, "On dynamic malware payloads aimed at programmable logic controllers," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, ser. HotSec, 2011, pp. 10–10.

[33] S. McLaughlin and P. McDaniel, "Sabot: specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*. ACM, 2012, pp. 439–449.

[34] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A trusted safety verifier for process controller code." in *Network and Distributed System Security Symposium (NDSS)*, vol. 14, 2014.

[35] O. Pavlovic and H.-D. Ehrich, "Model checking plc software written in function block diagram," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST. Washington, DC, USA: IEEE Computer Society, 2010, pp. 439–448. [Online]. Available: http://dx.doi.org/10.1109/ICST.2010.10

[36] O. Pavlovic, R. Pinger, and M. Kollmann, "Automated formal verification of plc programs written in il," in *Conference on Automated Deduction (CADE)*, 2007, pp. 152–163.

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

[38] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial internet of things," in *Proceedings of the 52nd annual design automation conference (DAC)*. ACM, 2015, p. 54.

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

[40] S. Senthivel, S. Dhungana, H. Yoo, I. Ahmed, and V. Roussev, "Denial of engineering operations attacks in industrial control systems," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY, 2018, pp. 319–329.

[41] R. Spenneberg, M. Brüggemann, and H. Schwartke, "Plc-blaster: A worm living solely in the plc," *Black Hat Asia*, 2016.

[42] H. Yoo and I. Ahmed, "Control logic injection attacks on industrial control systems," in *Proceedings of the 34th IFIP International Conference on Information Security and Privacy Protection*, 2019.

[43] S. Zonouz, J. Rrushi, and S. McLaughlin, "Detecting industrial control malware using automated plc code analytics," *IEEE Security & Privacy*, 2014.

APPENDIX A
APPENDIX

.

TABLE IV: The results of the password reset attack

Total block size(KB)	# of project files	Max size zipHash (bytes)	Lowest addr. of code block (hex)	Avg. # of write	Avg. time (sec)	Attack success rate
$6 \sim 7$	5	831	0xe088	3325	13.48	100%
$7 \sim 8$	19	1712	0xe08c	2943	11.89	100%
$8 \sim 9$	25	2261	0xe08c	2034	8.21	100%
>9	3	3103	0xe26c	1468	5.89	100%
Total	52	3103	0xe088	2458	9.93	100%