Control Logic Obfuscation Attack in Industrial Control Systems

Nauman Zubair*, Adeen Ayub†, Hyunguk Yoo*, Irfan Ahmed†
* Department of Computer Science, University of New Orleans, USA
† Department of Computer Science, Virginia Commonwealth University, USA
*{nzubair@my.uno.edu, hyoo1@uno.edu}, †{ayuba2, iahmed3}@vcu.edu

***Abstract*—Industrial control systems (ICS) are critical for safe and efficient operations of critical infrastructures such as power grids, pipelines, and water treatment facilities. Attackers target ICS, mainly programmable logic controllers (PLC), to sabotage underlying infrastructure. A PLC controls a physical process through connected sensors and actuators. It runs a control-logic program that specifies monitoring and controlling a physical process and is a common target of a cyberattack. A vendor-provided proprietary engineering software is typically used to investigate the infected control logic. This paper shows that an attacker can use control-logic obfuscation as an anti-forensics technique to hinder the investigations and incident response. The control-logic obfuscation subverts the engineering software's decompilation function; therefore, we call it a denial-of-decompilation (DoDe) attack. The DoDe attack exploits a fundamental design principle of creating compiled control logic in engineering software, thereby affecting the engineering software of multiple vendors in the industry.**

***Index Terms*—Control-logic attacks, digital forensics, industrial control system (ICS), programmable logic controller (PLC)**

I. Introduction

Industrial control systems (ICS) are parts of critical infrastructure that monitor and control physical processes such as power grids, oil and gas pipelines, and nuclear facilities. In a typical setting of ICS, programmable logic controllers (PLC) control a physical process at a field site, while supervisory applications at a control center are used to monitor the process and configure the PLCs over a network. Engineers at the control center use vendor-provided programming software, called *engineering software*, to program *control logic*, which defines how a PLC controls a physical process.

Previous security incidents on ICS (e.g., Stuxnet [1] and Triton [2]) and academic studies [3]–[6] have shown that the control logic of a PLC is vulnerable to malicious modification, called *control-logic attacks* [7]. In investigating control-logic attacks, the current state-of-the-practice heavily relies on the engineering software to obtain and analyze the control logic from a suspicious PLC. Especially, without the aid of the engineering software's decompilation function, it is hard to reveal the semantics of the attacker's control logic. Unlike the IT systems, the ICS environment largely involves proprietary network protocols, undocumented binary formats, and highly diversified designs and implementations, making it difficult for the digital forensic community to develop third-party tools to acquire and analyze forensic artifacts from ICS-specific embedded devices. In particular, ICS vendors typically employ proprietary compilers to translate control logic in a PLC programming language (defined in IEC 61131-3) into bytecode or machine code, generating a control-logic binary in an undocumented format, which may call proprietary library functions. These heterogeneous and closed nature of ICS make it difficult for forensic investigators to analyze the control logic directly in an assembly language or develop a third-party decompiler. Although Falliere [8] created the JEB decompiler for Siemens S7 PLCs and Keliris *et al.* [9] developed ICSREF for CODESYS, developing a third-party decompiler requires painstaking manual reverse-engineering for each PLC model or platform.

In this paper, we argue that the fundamental design principle of engineering software in compiling and decompiling control logic makes it vulnerable to *control-logic obfuscation*. The primary purpose of control-logic obfuscation in ICS is to delay the digital forensic investigations and incident response by subverting the engineering software's decompilation function; therefore, we call it the *denial-of-decompilation (DoDe) attack* in this paper. We evaluate the DoDe attack on two major PLC manufacturers' PLCs (i.e., Schneider Electric Modicon M221 and Siemens S7-300) and their engineering software (i.e., SoMachine Basic and TIA Portal). In addition, we empirically show that the DoDe attack can also evade a machine learning-based control-logic detection [10] that detects network packets containing control-logic binaries.

The rest of the paper is organized as follows. Section II provides the background and related work. Section III presents the DoDe attack, which exploits engineer software's design principle for compiling and decompiling control logic. Section IV shows the evaluation result, followed by Section V, where we discuss the implications of the attack and suggest defenses. Finally, Section VI concludes the paper.

II. Background and Related Work

A. Programmable Logic Controllers

Programmable logic controllers (PLC) are embedded systems used in various industries for automatic control of physical operations. A PLC has many input/output ports that can be wired with sensors and actuators to control various physical processes. For example, a single PLC can control the entire onsite generation of sodium hypochlorite at an offshore oil platform (to prevent the biofouling of intake pipes) with connected sensors (e.g., flow transmitters, level transmitters) and actuators (e.g., flow control valves, pumps).

The IEC 61131-3 standard defines five PLC programming languages: ladder logic, function block diagram, structured text, instruction list, and sequential function chart. Since these languages are domain-specific, they have specialized characteristics for the ICS domain. For example, a PLC program (or *control logic*) written in the ladder logic language is like a circuit diagram (as shown in Figure 2). PLCs can be programmed using a vendor-specific integrated development environment (IDE), commonly called *engineering software*. For example, Siemens PLCs are programmed with TIA Portal, Schneider Electric PLCs with SoMachine, Allen-Bradley PLCs with RsLogix 500 and Studio 5000 Logix Designer. Engineers at a control center use the engineering software to write and compile control logic, and then *download* it to a PLC over a network. The same software is also used to *upload* the control logic from the PLC and show the decompiled source code using its built-in decompiler.

B. Real-world Control-logic Attacks

Cyberattacks on PLCs may induce damage in the physical world. The attackers often aim to modify the control logic of a PLC, thereby negatively operating the underlying physical process. For example, Stuxnet [1] eventually downloads to target PLCs (Siemens S7-315 and S7-417) a malicious control logic that manipulates the rotor speed of centrifuges periodically from 1,410 Hz to 2 Hz to 1,064 Hz, which inflicted irreparable physical damage on about 1,000 centrifuges at Iran's nuclear facilities [1].

The Triton (a.k.a. Trisis or HatMan) malware [2], discovered at a Saudi Arabian petrochemical plant in 2017, targeted Schneider Electric Triconex Tricon 3008 controllers, which are designed to prevent hazards through a failsafe mechanism (e.g., safety shutdown) when abnormal conditions are present. The targeted controller type is a safety PLC that operates independently of a primary process control system, serving as a safety net for workers, machinery, and the environment when the primary control system fails. Therefore, the failure of a safety PLC can directly contribute to a catastrophic industrial disaster, including the loss of lives. The attack in 2017 [2] caused the targeted petrochemical plant to experience unexpected shutdowns triggered by infected controllers. Triton represents the first malware designed to attack industrial safety systems.

C. Use of Engineering Software in Forensic Investigation

The attacker's control logic codifies her malicious intention to the physical world; hence a high priority needs for understanding the control logic in the digital forensic investigation. In the current state-of-the-practice, engineering software plays a critical role in digital forensics and incident response against ICS cyberattacks. Unlike forensic investigations on typical IT systems, decompilation is often an essential step to understand the semantics of the attacker's control-logic binary since the the ICS environment is highly heterogeneous and exclusive. In ICS-specific embedded devices, their binary formats and the firmware are often proprietary. Moreover, the library/system functions called in the control-logic binaries and the information of a PLC's memory-mapped I/O are generally not-well documented. The investigators may consult with PLC manufacturers, but it will significantly delay response time, causing extended damage to the physical world.

The built-in decompiler of engineering software allows the forensic investigators examine the acquired control logic in a high-level PLC programming language (e.g., ladder logic, functional block diagram) that can be readable by the ICS engineers. According to the sources where the control logic is obtained, there are two approaches to analyze it. If the control logic is running on a PLC, the engineering software can directly upload the control logic from the PLC and decompile it. On the other hand, if it is acquired from a network/memory dump, the investigator can use a virtual-PLC (such as [3] and [11]) that replays the control logic in a conversation with engineering software.

D. Denial of Engineering Operation (DEO) Attacks

Senthivel *et al.* [12] defined *engineering operation* as "a continuous cycle of developing and updating the PLC control logic in response to changing operational requirements in ICS". They presented three attack scenarios, referred to as *denial-of-engineering* (DEO) attacks, in which an attacker can interfere with the normal engineering operation. The primary goal of the DEO attacks is to subvert engineering software's capabilities to acquire the actual control logic from an infected PLC.

The *denial-of-decompilation* (DoDe) attack (presented in Section III), which disables the engineering software's decompilation capability, can be considered as a specific method to implement the DEO attacks. More specifically, it can be used for the third DEO attack scenario where an attacker creates a well-crafted control logic that runs on a PLC successfully but crashes the engineering software when attempting to acquire the control logic from the PLC [12]. They demonstrated the attack by creating a malformed control-logic that exploits an inconsistency of input validation between engineering software and a PLC. In their study, Rockwell Automation's RSLogix 500 (the engineering software) refused to decompile a control-logic when integrity checks failed, while the Allen-Bradley MicroLogix 1400 PLC ran the logic successfully.

Their approach, however, needs to find a malformed instance that can trigger the discrepancy between engineering software's and the PLC's input validation, which is by no means a trivial task; in some cases, such inconsistency may not exist at all. In addition, PLC manufacturers can quickly fix the problem; likewise, investigators can easily correct the mismatched integrity-check values in a suspect control logic.

III. Denial-of-decompilation attack

This section proposes a new approach for the DEO attacks, referred to as the *denial-of-decompilation (DoDe) Attack*,

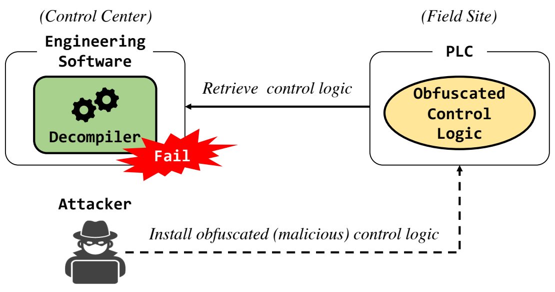


Fig. 1. An overview of the denial-of-decompilation (DoDe) attack in ICS

which exploits a fundamental design principle of engineering software's compilation/decompilation. Figure 1 shows an overview of the DoDe attack. An attacker installs an obfuscated control logic into a PLC and leaves the network. Then, when a forensics investigator attempts to acquire the control logic from the PLC using engineering software, the attempt fails since the software's decompilation function is not defined under the obfuscated control logic.

In existing control-logic attacks, an attacker prepares her malicious control logic using the engineering software of a target PLC—i.e., the attacker's control logic uses the same compilation choices of the legitimate engineering software. On the other hand, the DoDe attack subverts the software's decompilation through control-logic obfuscation that takes different compilation choices.

A. Compilation and Decompilation in Engineering Software

We can define a compilation as a multi-valued function τ, given the source language L_1 and the target language L_2. For each string (source program) $x \in L_1$,

$$\tau : L_1 \mapsto \mathcal{P}(L_2);\ \tau(x) = \{y \in L_2 : sem(x) = sem(y)\}$$

where $sem(x)$ represents the semantics of the program x. Note that the compilation is multi-valued since usually some statement of a high-level (source) language can have several different realizations in a low-level (target) language. However, if we consider a particular compiler f, the compilation is single-valued, because the compiler selects exactly one out of the many possible low-level implementations of the source program [13].

$$f : L_1 \mapsto L_2;\ f(x) = y \text{ s.t. } y \in L_2 \wedge sem(x) = sem(y)$$

In general, a compilation is not injective because more than one source programs may be translated into the same target program. Therefore, in typical IT systems, a decompilation (reconstructing a source program from a target program) is not the inverse of a compilation; compilation and decompilation are independently designed and do not necessarily make the same design decisions for their mappings [13].

On the other hand, the compilation in engineering software is, in general, injective—namely, two different control-logic source programs are always translated into two different target programs. There is an evident advantage in this design principle. Engineers at a control center often need to examine or debug the control logic running in a PLC, which requires the *original* source program. We can make this possible in two ways. The first way is to transfer the binary and source code together when updating the control logic of a PLC. However, this approach wastes limited PLC memory resources to store the source code. Moreover, the source code can be exposed during transmission since most ICS protocols do not support encryption. The second way is to make the compilation function *invertible*; given a compiler f, we define a decompiler g such that:

$$g : f(L_1) \mapsto L_1;\ g(f(x)) = x \text{ for all } x \in L_1$$

We exploit this design principle in compilation and decompilation to achieve our attack goal (i.e., install into a PLC a control logic that cannot be decompiled in engineering software). Since the domain of a decompiler g is restricted to $f(L_1)$, which is a strict subset of L_2, given a source program $x \in L_1$, we can find a target program y such that $y \in \tau(x)$ but $y \notin f(L_1)$. In other word, the target program y has the same semantics as the source program x, but it can never be generated by the particular compiler f. The function g cannot decompile the target program since it is not defined for y.

B. Control-logic Obfuscation

We can find such a target program y ($y \in \tau(x)$ but $y \notin f(L_1)$) through obfuscation, which is a common practice in malware development in the IT domain. We can define an obfuscation as a multi-valued function δ:

$$\delta : f(L_1) \mapsto \mathcal{P}(L_2);\ \delta(f(x)) = \{y : y \in \tau(x) \wedge y \neq f(x)\}$$

In other words, given a target program $f(x)$, which is the output of a particular compiler f on an input source program x, the obfuscation produces a set of morph y, which is a target program whose semantics is the same as $f(x)$ (and so as x), but whose realization is different. Note that a morph $y \in \delta(f(x))$ could be a member of $f(L_1)$ by chance, meaning y is defined under a decompiler g. Then, $g(y)$ will produce x' such that $sem(x) = sem(x')$ and $x \neq x'$ (x and x' cannot be the same since the compilation function is injective). We can test whether a morph can be used for the DoDe attack using the decompiler g (in practice, we use engineering software for the test). If the decompiler generates an error, we can use it, otherwise select another morph from $\delta(f(x))$.

To implement a particular obfuscator $\delta'(f(x)) \subseteq \delta(f(x))$, we can borrow common obfuscation strategies that have been extensively studied in the IT domain. However, the purposes are somewhat different. In the IT domain, attackers often obfuscate their malware greatly, and performance is a low priority. On the other hand, a PLC and its physical process operate within the real-time constraint; thus, complex obfuscation techniques (e.g., emulation-based, return-oriented programming), which can significantly increase the execution time, may not be suitable. We argue that simple obfuscation is enough for our purpose to hinder incident response due to the reasons mentioned in Section II—it is challenging to analyze control-logic binary even without obfuscation (due

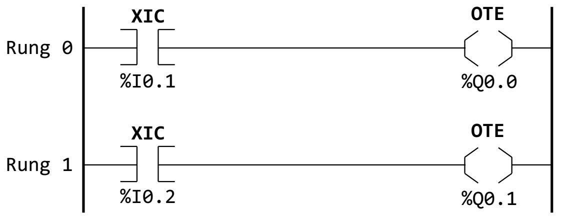


Fig. 2. An example control-logic source (Modicon M221)

to the heterogeneity and exclusiveness) if a decompiler is unusable. The following section presents two case studies that perform the DoDe attack through simple instruction-level obfuscation on two major manufacturers' PLCs.

IV. Experimental Evaluation

We have evaluated the DoDe attack on two different manufacturers' PLCs: Schneider Electric Modicon M221 and Siemens S7-300. We utilized the well-known instruction-level obfuscation strategies such as garbage-code insertion, equivalent-instructions substitutions [14]. Given a control-logic source program, we first compiled it using engineering software, and extracted the control-logic binary. Then, we disassembled the binary into an assembly code to which obfuscation was applied, and assembled back to machine code. The obfuscated control-logic binary was transferred and installed into a PLC, then we checked that the control logic ran successfully in the PLC while engineering software failed to decompile when attempting to acquire the control logic from the PLC. In addition, although it is not the primary goal of the DoDe attack, we conducted a separate experiment to see whether the obfuscated control logic can also evade `Shade` [10] which is a ML-based control-logic detection.

A. Subverting Decompilation in Engineering Software

Attack on Modicon M221 PLC. Our first subject were the Schneider Electric Modicon M221 PLC (firmware v1.6.0.1) which runs on a Renesas RX630 microcontroller, and SoMachine Basic (v1.6), the engineering software. We utilized a toolchain[1] provided by Renesas to perform assembling/disassembling for the RX architecture. The obfuscated control logic was transferred into the PLC using a rouge clients [4].

Figure 2 shows a control-logic source program. The `XIC` (examine-if-closed) instruction on the first line (Rung 0) examines a PLC's digital input `%I0.1` (input port 1 on slot 0), and if the bit is set, then the connected `OTE` instruction is executed, which sets the digital output `%Q0.0` (output port 0 on slot 0). Then, the next line (Rung 1) is executed in a similar way, and then over again for each scan cycle.

Table I represents the original target program (produced by SoMachine Basic) for the example source program, and its morphs generated through instruction-level obfuscations. In the original program, `BTST #1,r12` examines bit 1 (i.e., the second to the LSB) of the `r12` register which reflects the digital inputs of the PLC; namely, it tests `%I0.1`. The `BTST` instruction has two operands—`BTST src,src2`. If `src2` is a register, then it sets the carry flag as following:

$$Carry\ flag\ =\ ((\ src2\ >>\ (\ src\ \&\ 31\ ))\ \&\ 1\ )$$

Thus, `BTST #1,r12` sets the carry flag only if bit 1 of `r12` is set. The next instruction `BMC #0,r13` sets bit 0 of `r13` if the carry flag is set. Since the bits of `r13` are mapped to the PLC's digital outputs, the instruction basically actuates the output device connected to the output port 0 on the PLC's slot 0, when the carry flag is set, and vice versa. Lastly, `RTS` is a return instruction.

Morph 1—It was generated through equivalent-instructions substitution. The `TST` instruction, replacing `BTST` in the original code, performs a logical AND operation on its two operands and sets the zero flag if the result is zero, otherwise clears. Then, we can substitute `BMC` with `BMNE` which sets a bit if the zero flag is 0, otherwise clears the bit. To sum up, an instruction sequence (`BTST`, `BMC`) can be substituted with an equivalent sequence (`TST`, `BMNE`) in the RX machine code.

Morph 2—It represents a bit more complicated obfuscation. First, `MOV.L #0, r1` clears `r1` because we will use `r1` later. Then, the next `BTST` instruction checks bit 1 of the input register (`r12`) and modifies the carry flag accordingly as in the original program. However, the logic executed by a single `BMC` instruction in the original program was stretched over three instructions—`BMC`, `CMP`, and `BMGE`.

Both morphs ran successfully on the Modicon M221 PLC, but SoMachine Basic failed to decompile them.

TABLE I
Original and obfuscated programs (M221)

Original	Morph-1	Morph-2

`BTST #1, r12`
`BMC  #0, r13`
`BTST #2, r12`
`BMC  #1, r13`
`RTS`
	`TST  #2, r12`
	`BMNE #0, r13`
	`TST  #4, r12`
	`BMNE #1, r13`
	`RTS`
		`MOV.L #0, r1`
		`BTST  #1, r12`
		`BMC   #1, r1`
		`CMP   #1, r1`
		`BMGE  #0, r13`
		`MOV.L #0, r1`
		`BTST  #2, r12`
		`BMC   #1, r1`
		`CMP   #1, r1`
		`BMGE  #1, r13`
		`RTS`

Attack on S7-300 PLC. Our second subject were Siemens S7-300 (firmware v3.2.17) and TIA Portal (v16). We used Radare2 with a library[2] to disassemble MC7 bytecode (which is the target language for S7-300) into the STL language (which is an assembly-like language corresponding to MC7 bytecode). The Snap7 library and its python wrapper[3] were used to download the obfuscated code into the PLC.

[1]https://gcc-renesas.com/

[2]https://github.com/wargio/libmc7

[3]https://github.com/gijzelaerr/python-snap7

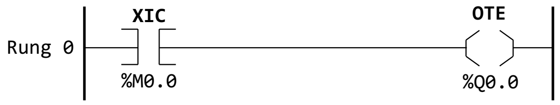
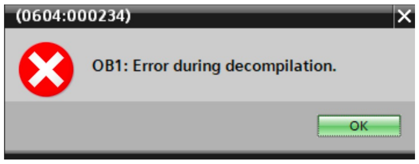


Fig. 3. An example control-logic source (S7-300)

TABLE II
ORIGINAL AND OBFUSCATED PROGRAMS (S7-300)

Original			Morph-1		
Offset	MC7	STL	Offset	MC7	STL

`0x24`
`0x26`
`0x28`
	`8000`
	`d880`
	`6500`
		`A M 0.0`
		`= Q 0.0`
		`BE`
			`0x24`
			`0x28`
			`0x2a`
			`0x2c`
				`700b00`
				`02`
				`8000`
				`d880`
				`6500`
					`JU 0x28`
					`A M 0.0`
					`= Q 0.0`
					`BE`


(0604:000234)

OB1: Error during decompilation.

OK

Fig. 4. An decompilation error message from TIA Portal

Figure 3 shows an example control-logic source. Table II describes the original target program (produced by TIA Portal) for the example source program, and a morph generated through garbage-code insertion. In this example, the garbage code is the jump-unconditional (JU) instruction that we used for merely jumping to the next instruction; namely, it plays like a no-operation (NOP) instruction. The obfuscated code ran well on S7-300 while TIA Portal generated an error message (refer to Figure 4) when attempting to retrieve the control logic from the PLC.

B. Evading ML-based Control-logic Detection

We also evaluated the DoDe attack against `Shade` [10], which detects network packets containing control-logic code using a ML-based approach. Since `Shade` does not support Siemens S7-300, we conducted an experiment only for Modicon M221.

Dataset. We generate a dataset that contains 14 original and obfuscated control logic. To generate the dataset, we first program 14 different control logic using SoMachine Basic, then captures the downloading traffic of control logic to the PLC, from which the binary control logic is extracted. After disassembling the binary logic, we apply two obfuscation strategies on the assembly code level; 1) inserting `NOP` instructions between each assembly instruction; 2) substituting (`TST`, `BMNE`) for (`BTST`, `BMC`).

Detection result. We configured `Shade` using a support-vector machine (SVM) with the radial basis function (RBF) kernel. Table III shows the detection accuracy of control logic code in `Shade` at 1% false postive rate. Among the 42 different features studied in [10], we select the *#dec* feature (the byte size of decompiled logic code[5]) and the *L4gram* feature (the longest continuous match of 4-grams that are present in a pre-generate bloom filter[6]) for a SVM classifier used in `Shade`, which showed the best detection rate in [10]. When `Shade` uses only the *#dec* feature, it did not detect a single packet which contains logic code among 1700 code packets when the code is obfuscated, while the detection rate is 83.16% without obfuscation. When the *L4gram* feature is used, `Shade` detects obfuscated code slightly better, but almost meaningless (3.59%). If we consider `Shade` to detect control logic when it detects at least one of the packets containing the logic code, the detection rate is 100% on the original code while it drops to 21.43% at best on the obfuscated code.

TABLE III
DETECTION RATE AT 1% FALSE POSITIVE RATE

	Original	Obfuscated
# of control logic	14	14
# of write request packets (*all*)	1284	2420
# of write request packets (*code*)	578	1700
control logic detection - feature set: {*#dec*}	14/14 (100%)	0/14 (0%)
control logic detection - feature set: {*L4gram*}	14/14 (100%)	3/14 (21.43%)
control logic detection - feature set: {*#dec, L4gram*}	14/14 (100%)	3/14 (21.43%)
code packet detection - feature set: {*#dec*}	469/578 (83.16%)	0/1700 (0%)
code packet detection - feature set: {*L4gram*}	486/578 (86.17%)	61/1700 (3.59%)
code packet detection - feature set: {*#dec, L4gram*}	522/578 (92.6%)	63/1700 (3.71%)

The experiment result was somewhat expected because the training dataset of `Shade` was collected from the normal control logic, which was not obfuscated. This result indicates that the DoDe attack may also evade the machine-learning models trained using the compilation output produced by engineering software.

V. DISCUSSION

Implication. Code obfuscation is widely used in the development of both benign or malicious software in the IT domain. An attacker often obfuscates her malicious code to impede the defender's reverse engineering process. Deobfuscation techniques have been proposed [15], [16] to simplify the obfuscated code while preserving its semantics. However, without knowing the attacker's choice of obfuscation, it is very unlikely, if not impossible, to recover the *original* code. In digital forensics for PLCs, where the investigator heavily relies on the engineering software, only the original code, not just simplified code, is required to make it decompiled and show the semantics in the context of the underlying physical process.

[5]`Shade` uses `Eupheus` to decompile the logic code of the M221 PLC.

[6]The bloom filter is generated from 22 binary control logic compiled by SoMachine Basic.

In reality, an attacker may employ a man-in-the-middle (MITM) attack along with a control-logic attack, like Stuxnet, to prevent the defender from acquiring the malicious control logic at all, making them get a copy of a benign program. However, once the defenders get rid of the MITM, she can obtain the malicious control logic and feed it to the engineering software to see its source code. The DoDe attack can be used together with a Stuxnet-style attack, to further hinder the forensic investigation and incident response. Although we showed only a few obfuscations in Section IV just for demonstration purpose, there are many possible obfuscations that the attacker can choose, and therefore it is difficult for the defender to learn the implementation details for the attacker's obfuscation and deobfuscate it to the original form that can be decompiled by the engineering software.

Mitigation. PLCs generally support three different operation modes, which can be configured by a physical method (e.g., hardware key). In the run mode, one can not overwrite the control logic in a PLC's memory. In the program mode, a PLC can be programmed by engineering software. And in the remote mode, engineers can remotely change the mode of the PLC. One of the best practices to keep the PLC protected is to operate the PLC in the run mode during regular operation. However, in practice, PLCs often run in the remote mode for convenience. Note that a PLC without further memory protection can still be vulnerable to data execution attacks [4] since the run mode does not prevent an attacker from injecting code into the data section in the PLC's memory.

Code signing is a mechanism of authenticating the authors of executables based on cryptographic measures. It is widely used in IT systems to authenticate the code publisher and provide the integrity of the code. Cryptographic hardware modules such as HSM can be used to safely keep the private key. However, to the best of our knowledge, code signing is not used in PLCs yet. Although a code signing system can be breached if not correctly designed [17], it can improve the security of PLCs against control-logic attacks in general.

VI. Conclusion

In this paper, we discuss the limitation of using engineering software as a digital forensic tool to investigate control-logic attacks. We present the denial-of-decompilation (DoDe) attack, which exploits engineering software's fundamental design principle in compiling and decompiling the control logic of a PLC, which makes the attack generally applicable. We have evaluated the attack on two major vendors' PLCs. Experiments using instruction-level obfuscation have shown that the attack is effective. In addition, we also shows that the DoDe attack may also be effective to evade ML-based control-logic detection. Based on the findings in this study, we argue that current ICS forensic capabilities relying on engineering software are incomplete, and the ICS community needs to develop more robust strategies and tools to respond sophisticated control-logic attacks which involve different techniques including control-logic obfuscation.

Acknowledgment

This material is based upon work supported by the U.S. Department of Homeland Security under Grant Award Number 17STCIN00001-05-00. In addition, this work has received funding from the Louisiana Board of Regents' Research Competitiveness Subprogram (RCS) under contract number LEQSF(2021-22)-RD-A-27.

Disclaimer. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security.

References

[1] N. Falliere, L. O. Murchu, and E. Chien, "W32.stuxnet dossier version 1.4," Symantec, Tech. Rep. 6, 2011.

[2] A. Di Pinto, Y. Dragoni, and A. Carcano, "TRITON: The first ICS cyber attack on safety instrument systems," in *Proceedings of 2018 Black Hat USA*, 2018.

[3] S. Kalle, N. Ameen, H. Yoo, and I. Ahmed, "CLIK on PLCs! Attacking Control Logic with Decompilation and Virtual PLC," in *Proceedings of 2019 NDSS Workshop on Binary Analysis Research (BAR)*, 2019.

[4] H. Yoo and I. Ahmed, "Control logic injection attacks on industrial control systems," in *Proceedings of the 34th IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*. Springer, 2019, pp. 33–48.

[5] S. E. McLaughlin, "On dynamic malware payloads aimed at programmable logic controllers," in *Proceedings of the 6th Usenix Workshp on Hot Topics in Security (HotSec)*, 2011.

[6] S. McLaughlin and P. McDaniel, "SABOT: Specification-based payload generation for programmable logic controllers," in *Proceedings of 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 439–449.

[7] R. Sun, A. Mera, L. Lu, and D. Choffnes, "Sok: Attacks on industrial control logic and formal verification-based defenses," in *Proceedings of 2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021.

[8] "JEB Decompiler for Siemens S7 PLC," https://www.pnfsoftware.com/jeb/plc, [Online; accessed 19-April-2022].

[9] A. Keliris and M. Maniatakos, "ICSREF: A framework for automated reverse engineering of industrial control systems binaries," in *Proceedings of 2019 Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[10] H. Yoo, S. Kalle, J. Smith, and I. Ahmed, "Overshadow PLC to detect remote control-logic injection attacks," in *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2019, pp. 109–132.

[11] S. A. Qasim, J. M. Smith, and I. Ahmed, "Control logic forensics framework using built-in decompiler of engineering software in industrial control systems," *Forensic Science International: Digital Investigation*, vol. 33, p. 301013, 2020.

[12] S. Senthivel, S. Dhungana, H. Yoo, I. Ahmed, and V. Roussev, "Denial of engineering operations attacks in industrial control systems," in *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018, pp. 319–329.

[13] S. C. Reghizzi, L. Breveglieri, and A. Morzenti, *Formal languages and compilation*. Springer, 2013.

[14] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[15] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

[16] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 275–284.

[17] D. Kim, B. J. Kwon, and T. Dumitraş, "Certified malware: Measuring breaches of trust in the windows code-signing pki," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1435–1448.