Review of ICSREF: A Framwork for Automated Reverse Engineering of Industrial Control Systems Binaries

Zachary Norvell
Electrical and Computer Engineering
Texas A&M University
College Station, TX, USA
norvezac000@tamu.edu

Rhys Vennema
Electrical and Computer Engineering
Texas A&M University
College Station, TX, USA
rhysvennema@tamu.edu

***Abstract*—The convergence of operational technology (OT) and information technology (IT) networks has improved efficiency and reliability in industrial control systems (ICS), but it has also introduced new cybersecurity risks that have caused large disruptions and losses while evading detection. An important component of ICSs are programmable logic controllers (PLCs), which serve as the primary interface between control logic and physical processes. Because PLCs were originally designed for isolated networks, they often lack modern cybersecurity protections, making them vulnerable when connected to integrated environments. Understanding and analyzing PLC binaries is therefore essential for detecting and mitigating malicious code that targets these systems. The work reviewed in this paper presents an automated framework for reverse engineering PLC binaries, aiming to accelerate the identification of malicious modifications and improve the security of ICS environments.**

***Index Terms*—Industrial Control Systems (ICS), Programmable Logic Controllers (PLCs), cybersecurity, reverse engineering, PLC binaries, CODESYS, Control Flow Graph (CFG)**

I. Motivation of the original paper

Industrial control systems (ICS) control physical processes in industrial sectors and critical infrastructures so we can achieve desired outputs from given inputs. An ICS controls many individual components such as programmable logic controllers (PLCs), control loops, human-machine interfaces (HMIs), and remote diagnostic and maintenance tools [3].

Conventionally, industrial control systems have been implemented on isolated networks, but business needs have driven companies to converge ICS operational technology (OT) with information technology (IT) networks. The convergence between the two has allowed for improved control, efficiency, and reliability of ICSs, but it has also led to an increase in cybersecurity threats [1][3]. ICS components, which have been designed for isolated networks, now face cybersecurity risks of integrated networks and real world attacks have taken advantage of this interconnectivity to attack vulnerable ICS components, resulting in major disruptions and losses.

A famous example of an ICS attack was Stuxnet which setback Iran's nuclear enrichment facility in 2010. The attack utilized malware that spread across networks undetected and was capable of replacing compiled PLC blocks for Siemens PLCs with malicious binaries which changed physical processes, such as the speeds of Iran's centrifuges, while outputting normal values to the supervisory control & data acquisition (SCADA) system [5]. This attack demonstrates the severity of this ICS cybersecurity threat and it is important to find methods to mitigate attacks utilizing malicious PLC binaries.

As we review [1] we will discuss the automated reverse engineered framework which aims to automate reverse engineering of PLC binaries. By automating this task, which usually requires long manual analysis of proprietary PLC binaries, the method could be used to accelerate detection of malicious code before it can disrupt an ICS. Consequently, this automated framework can then be used by attackers, improving their knowledge of an ICS environment to facilitate more advanced ICS attacks.

II. Background

In this section we provide background information on ICS and PLC environments to help understand the methods used in [1] in order to automate reverse engineering of PLC binaries.

A. Industrial Control Systems

ICSs control physical processes in industrial sectors and critical infrastructures. ICS is a term used for a wide variety of control systems. Supervisory Control and Data Acquisition (SCADA) monitor ICS components that are geographically remote, collecting data and issuing commands to the remote stations [3]. Distributed control systems (DCS) are used to describe the distributed control elements across an industrial plant which are used to automate a process. The automation of processes in ICSs relies on these automated controls, environment sensors, and PLCs in order to make decisions and act on changes in an ICS in real time in order to achieve the desired outputs from given inputs and to better control efficiency, reliability, and quality.

B. Programmable Logic Controllers

PLCs are embedded systems that are made for industrial environments. They typically are built for reliability and can

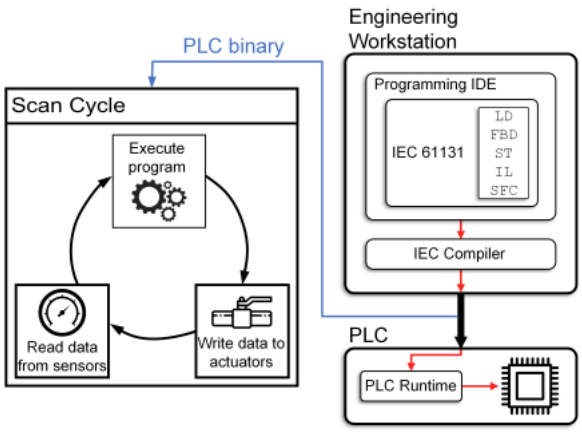


Fig. 1. PLC scan cycle and software development process. Reproduced from [1]

last from 10 to 30 years depending on conditions. PLCs have been optimized to efficiently work in isolated networks and they infinitely execute what is known as a scan cycle which has three major components and is illustrated in Fig. 1. PLCs are programmed using vendor provided Integrated Development Environments (IDEs) which correspond to specific PLC models. This is also illustrated in Fig. 1. After the PLC logic has been programmed by a process engineer, the PLC logic is compiled using the vendor's proprietary PLC compiler, and it can then be downloaded to the PLC for execution. The PLC binaries are stored in its nonvolatile memory and are loaded to volatile memory on startup for faster execution.

C. PLC Binaries

PLC binaries vary from those of conventional workloads due to the repeating nature of the scan cycle workload. Unlike analysis of regular terminating programs, analysis of ICSs require storage of useful information for later analysis in the event that an issue in the process was found. Another variation from PLC workloads from conventional ones is that PLCs rely more on I/O operations, taking up two thirds of the scan cycle [1]. Since I/O operations allow PLCs to interact with physical processes and communicate with an ICS environment, it is important to identify how PLC binaries orchestrate I/O operations. Since PLCs rely on proprietary compilers, it is also important to understand the file format the compiler uses. The compiler that [1] focused on automating reverse engineering was the CODESYS compiler, the reason for this being the usage of CODESYS across multiple PLC providers, which is illustrated in Table I. However, reverse engineering can be done on other compilers using the same principles that [1] discussed. A final component of PLC binaries is how the compilers handle optimizations. Due to the criticality of PLCs, optimizations have to be conservative. This is different compared to binaries of noncritical machines but this helps the reverse engineering framework as binaries are simpler to understand.

TABLE I
AUTOMATION PLATFORMS OF ICS VENDORS REPRODUCED FROM [1]

Company	Development platform	CODESYS-based?
Rockwell Automation	Studio 5000 Logix Designer	No
Siemens	STEP7	No
ABB	Automation Builder	Yes
Schneider Electric	SoMachine	Yes
Bosch Rexroth	Indralogic	Yes
Wago Kontakttechnik	WAGO-I/O-PRO	Yes
Eaton Industries	XSOFT-CODESYS	Yes
Beckhoff Automation	TwinCAT	Yes
Lenze Automation	PLC Designer	Yes
Owen	CODESYS	Yes
Omron	CX-One	No
SEL	acSELerator	Yes
ifm electronic	CODESYS	Yes
STW Technic	CODESYS	Yes
Berghof Automation	CODESYS	Yes

III. METHODOLOGY

In this paper the ICSREF framework is presented as an automated process for reverse engineering PLC Binaries, identifying function blocks and library calls, constructing a control flow graph (CFG), and performing modifications for experimentation and attacks [1]. There were two phases proposed to carry out these tasks. First, a platform-specific phase gathers knowledge about each binary framework, and then a binary analysis phase, which uses the knowledge databases to break down a given binary to extract and reconstruct valuable information.

A. First Phase

The first phase, which came at a one time cost with recurring updates, was gathering information for each specific platform that may be used. Because there are many different PLCs and development environments, there are different formats in which the binaries can be compiled into. Some common PLC platforms include STEP7, Studio 5000, and CODESYS and PLC vendors include WAGO, OWEN, LENZE, and BERGHOF. Two knowledge databases were created to store extracted information by reverse engineering the different binaries and determining header contents, subroutine delimiters, subroutines, symbol tables and dynamically linked functions, and details on code and data sections. The first includes read and write information relating to I/O peripherals and corresponding memory addresses. The second database contains hashed signatures of known library functions and common code snippets used by the specific platforms. This includes common libraries used for network communication, process control, and timing. To make these easier to identify later on, the database holds a hashed version of these function blocks. By only using the specific series of opcodes of functions, a unique signature could be saved and used to search them in the database. To fill both of these databases, real programs were collected from public repositories on GitHub. These binaries vary in platform, authors, intended PLC types, and industrial environments they were created for. In total, 471

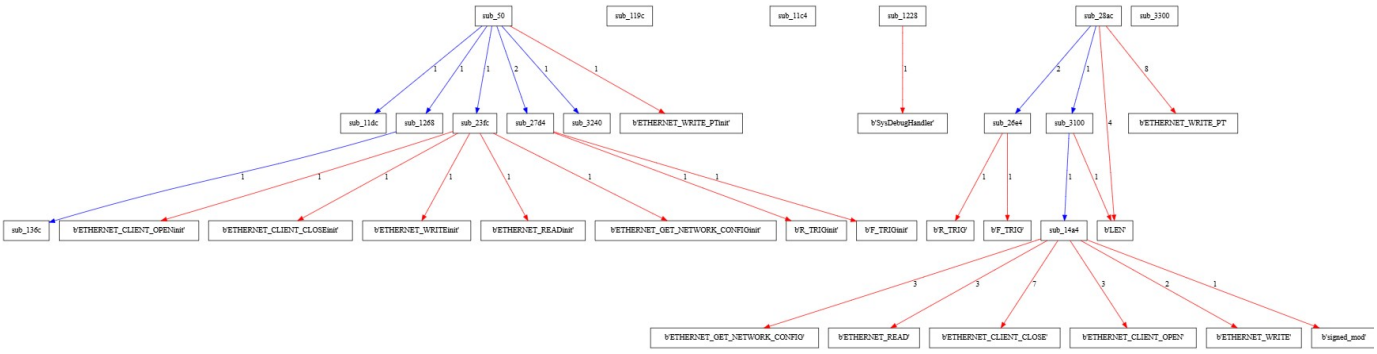


Fig. 2. Control Flow Graph generated by ICSREF.

project files were downloaded and 266 binaries were included and processed to form the databases that are used now.

B. Second Phase

With the first phase in place, a binary's platform can be identified and used to further analyze and reverse engineer. That brings us to the core of the ICSREF framework that is performed on each binary to disassemble it by rebuilding the assembly code and create a CFG detailing the structure of the entire program [1]. In order to get the information to use the databases, the process starts by parsing header information, carving out subroutines using RADARE2, and extracting the symbol table that contains information about variable names, function names, and their memory addresses. The next and most important step for truly reconstructing the binaries is creating the control flow graph (CFG). After each function is identified, they are then linked to each other to form the entire program flow. This includes dynamically linked functions from outside libraries that are also contained in the database. With the CFG in place, the databases can be used again to identify specific I/O information about how the PLC interacts with its environment and the purpose it serves. The function blocks that are matched with the database have the option to be further analyzed to identify arguments that are passed in and understand relationships between variables. This is crucial in making the ICSREF framework useful. After completion of binary reverse engineering, the program gives users the option to modify sections of the reconstructed binaries. With knowledge about function arguments and variables and their purpose in a program, users can alter these to change important output values such as operating points in a PLC.

C. Implementation

In a case study that the authors presented, they simulated an attack on a chemical process with a phone that contained the ICSREF program that would automatically run once connected to the PLC. After it ran, they generated a payload that modified a function input that had been identified in the binary reverse engineering process. In the context of the chemical process, the value they decided to alter was a pressure setpoint (Ka) in which they dropped to a value way off from its original value. After the injection of the payload, they were able to see the pressure begin to stabilize at the new value they set. They emphasize that a small change such as this may not have any drastic effect, but can go undetected and result in monetary loss for the victim organization. The experiment gave a good example of how this framework could be deployed and automated to do harm to a standard ICS.

This entire process was documented and programmed to run on any linux system. The project has a GitHub that anyone can clone and begin using or modify. Using Python, they developed a ICSREF shell to provide the user commands for processing imported binaries. The commands allow the user to analyze a certain binary in the directory, construct the CFG, hash and identify functions, run direct bash commands, and more tools. In the following section, we will run a sample binary that they had collected and see what results are produced.

IV. Implementation Details

Our analysis of the CODESYS framework was conducted locally using a Windows 11 system equipped with an Intel Core i7-11370H and 32 GB of DDR4 RAM. This configuration provided sufficient computational resources for running the project locally.

The framework requires the software to run on Ubuntu 22.04 LTS. We used WSL2 to run the necessary environment. We then followed the installation process documented in the CODESYS framework GitHub repository which notably requires Python 2.7.

We also tested implementation of the CODESYS framework using Python 3 as a recent branch of the repository was created in order to make the framework compatible with the updated Python version. We found this repository branch and installation methods also worked, so those looking to implement the framework locally can use either Python 2.7 or Python 3.

V. Full Results

After cloning and installing the framework using Python 2.7 we arrived at the terminal shown in Fig. 3. The shell created by the program provided a list of useful commands for analyzing the binaries. The project repository contains over 200 samples that we could choose from; however, after running the analysis

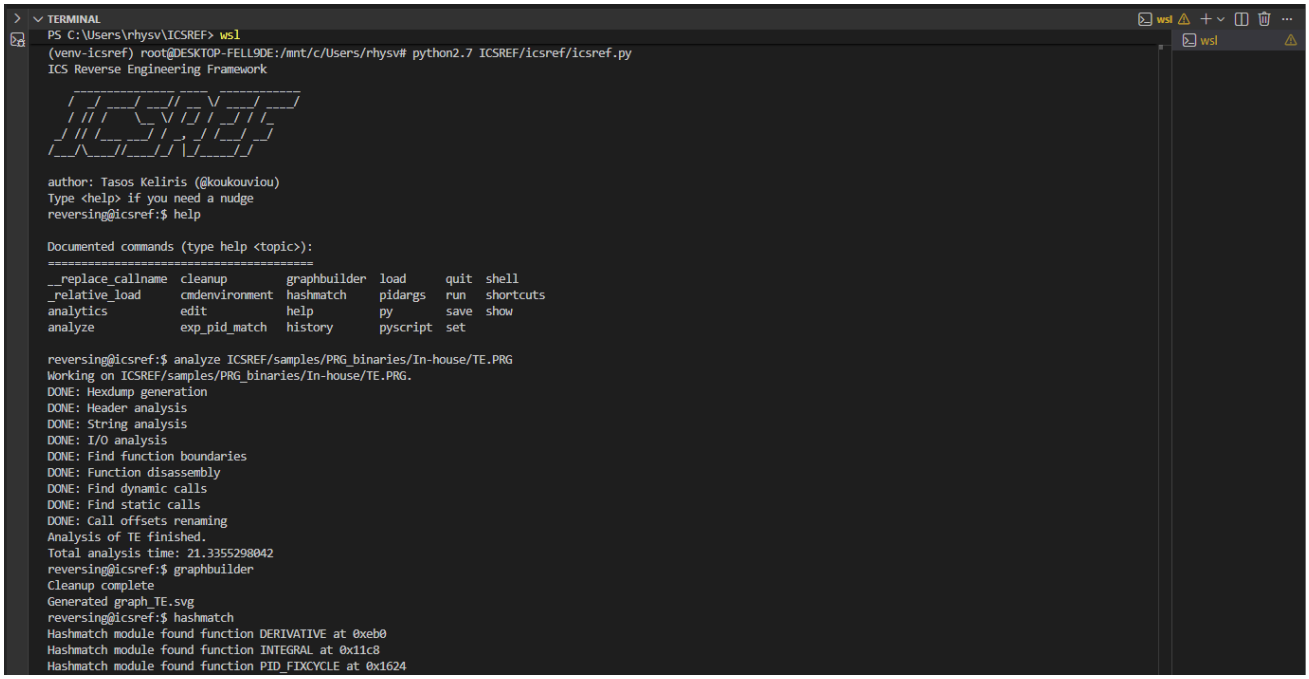


Fig. 3. ICSREF modules and analysis of in house PRG.

and graph builder on many of them, we quickly realized how large some of these disassemblies were. To get a good example CFG as seen in Fig. 2, we chose sample binary 25.PRG and used the analyze command as shown in the tutorial the authors provided. After generating the disassemblies, the graphbuilder command was called using the outputs of the analysis creating the CFG. By looking at the CFG it is clear that this program consisted of network I/O involving ethernet writes and other functions. Among the generated CFG were many other files that showed each subroutine disassembled as seen in Fig. 5, as well as an analytics file that identified known functions such as TRIG and ETHERNET related calls as well as how many times they were called. Using another binary we analyzed is shown in Fig 4. where the pidargs and analytics commands were used. These pidargs give vital variables used in the program as well as the addresses they were located. The analysis give the different functions and the calls they made. The disasm files that were created contained the different functions that were identified in assembly form. These were the snipped that were matched in the knowledge databases and identified using their specific sequence of opcodes.

Just by using a view commands the binaries could be completely broken down for an in depth analysis. ICSREF yielded many results that could be used by attackers or engineers that intend to improve the security of the way these binaries are compiled, how they interact with the PLC, and how they can be developed to make the process of reverse engineering binaries more difficult. To compare our results directly to [1] we ran the framework's modules on the in house PLC binaries that were used in the preview of the project. The preview is available to view in the framework's documentation, which can be used to compare our results to those of the authors. Like the demonstration given in the paper we were able to fully reverse engineer binaries and generate files that essentially recreated the original program and its structure. Unlike their case study we were unable to generate a malicious attack in an automated fashion using the outputs that were created.

VI. Insights

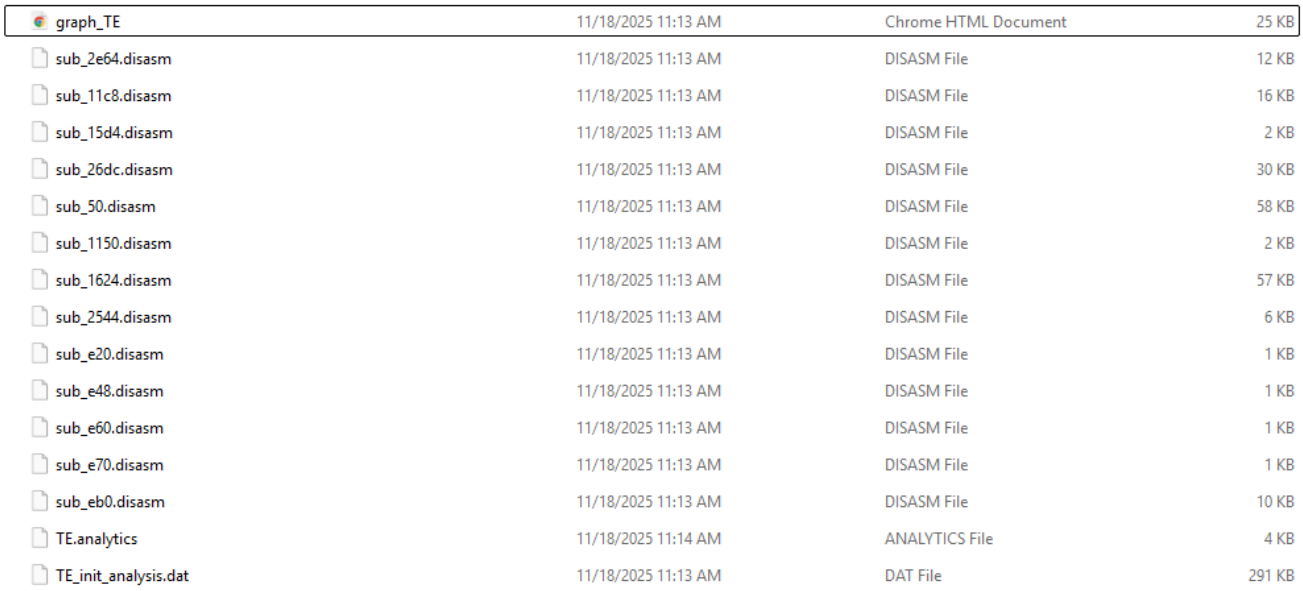
The version of the CODESYS framework cloned from the GitHub repository does not include the changepid module, a component shown in the project preview and a method of attacking that was discussed in [1]. We hypothesize that this module was removed as it provides malicious users with a quick and directed mechanism to attack ICS environments. As shown in Fig. 3, this module is absent from the available code base. Although the repository does not provide an explanation, it is reasonable that the omission was intentional, potentially due to concerns of misuse. All other modules which were shown in the preview were included with the project that we cloned, and comparing the results of the preview to the artifacts that we replicated. We believe that our local version of the framework is running as expected, as we were still able to replicate the first half of the framework by reverse engineering different binaries and getting their reconstructed disassemblies.

VII. Discussion

Given that this framework can contribute significantly to the development of ICS security as well as be used to maliciously attack PLCs, the full intellectual property of the ICSREF framework was not provided in the GitHub limiting

```
PS C:\Users\rhysv\ICSREF> wsl
reversing@icsref:$ exp_pid_match
reversing@icsref:$ pidargs

Call to PID at 0x2cd8
SET_POINT = 2800.0              (0x452f0000L)
       KP = -9.99999974738e-05  (0xb8d1b717L)
       TN = 0.333299994469      (0x3eaaa64cL)
       TN = 0.0                 (0x0)
 Y_MANUAL = 0.0                 (0x0)
 Y_OFFSET = 2.76036422764e+20   (0x616f6c66L)
    Y_MIN = 0.0                 (0x0)
    Y_MAX = 100.0               (0x42c80000L)
   MANUAL = 2.75550034004e+20   (0x616f0066L)
    RESET = 6.97683179758e+28   (0x6f616f00L)
    CYCLE = 0.000500000023749   (0x3a03126fL)

Call to PID at 0x2dc4
SET_POINT = 1.34081365819e-38   (0x92006eL)
       KP = 0.00999999977648    (0x3c23d70aL)
       TN = 1.66999998328e-05   (0x378c16faL)
       TN = 0.0                 (0x0)
 Y_MANUAL = 24.3999996185       (0x41c33333L)
 Y_OFFSET = 7.70105234979e+31   (0x7473009bL)
    Y_MIN = 0.0                 (0x0)
    Y_MAX = 100.0               (0x42c80000L)
   MANUAL = 2.75550034004e+20   (0x616f0066L)
    RESET = 6.97683179758e+28   (0x6f616f00L)
    CYCLE = 0.000500000023749   (0x3a03126fL)
reversing@icsref:$ analytics
GLOBAL_INIT --|1|--> SUB_2 <=> e2466609298d61443fe498dc2b47dbfa37a06f34de6fb3136a4856fa2dd729f5 --|1|--> 8aabf99b2d021bd68a716d40545777fd22c469d97a107c6c7e1face2213c21a1
GLOBAL_INIT --|4|--> PID_FIXCYCLE_INIT <=> e2466609298d61443fe498dc2b47dbfa37a06f34de6fb3136a4856fa2dd729f5 --|4|--> 6272173c4637da0257fb72e6d6b1ae85f9844ae88eb7fb48eef44436e3b7af7f
GLOBAL_INIT --|2|--> R_TRIGinit

SYSDEBUG --|1|--> SysDebugHandler

DERIVATIVE --|1|--> real_add
DERIVATIVE --|1|--> ulong_to_real
DERIVATIVE --|2|--> real_sub
DERIVATIVE --|1|--> real_div
DERIVATIVE --|2|--> real_mul
INTEGRAL --|2|--> real_sub
INTEGRAL --|1|--> real_lt
INTEGRAL --|3|--> real_mul

PID_FIXCYCLE --|2|--> real_eq
PID_FIXCYCLE --|4|--> real_div
PID_FIXCYCLE --|1|--> float_to_dword
PID_FIXCYCLE --|4|--> real_ne
PID_FIXCYCLE --|7|--> real_add
PID_FIXCYCLE --|5|--> real_gt
PID_FIXCYCLE --|4|--> INTEGRAL <=> 113d173c5712ef3fa03065f44b109516cabc7af102bf25f590c43983def3f34d --|4|--> 56a027c1dd7b107117764a3bc92c523095cb12a58dbddc5da910eda74e7619f3
PID_FIXCYCLE --|6|--> real_sub
PID_FIXCYCLE --|2|--> real_lt
PID_FIXCYCLE --|2|--> DERIVATIVE <=> 113d173c5712ef3fa03065f44b109516cabc7af102bf25f590c43983def3f34d --|2|--> bf1d54d93d1bebfff20c45eb889b81c8ffe7ade0e10cf3d3a9419a7a4f7b3116
PID_FIXCYCLE --|8|--> real_mul

PID_FIXCYCLE_INIT --|1|--> INTEGRAL_INIT <=> 6272173c4637da0257fb72e6d6b1ae85f9844ae88eb7fb48eef44436e3b7af7f --|1|--> 212608709946d8bc4c358a104cfd97642d83ed5f81c5efba06f1a2de12cfcdbd
PID_FIXCYCLE_INIT --|1|--> DERIVATIVE_INIT <=> 6272173c4637da0257fb72e6d6b1ae85f9844ae88eb7fb48eef44436e3b7af7f --|1|--> c859bbdedfbfc39a1a21860da25b647049137fd6f83a79dd7dd324f5c69d7d7b

PLC_PRG --|3|--> ulong_to_real
PLC_PRG --|6|--> real_div
PLC_PRG --|3|--> real_add
PLC_PRG --|3|--> real_sub
PLC_PRG --|1|--> real_to_long
PLC_PRG --|2|--> PID_FIXCYCLE <=> 159fd1a86b93cc89c4f4453a920882fccca9798c6c01618aa65d173866b533b1 --|2|--> 113d173c5712ef3fa03065f44b109516cabc7af102bf25f590c43983def3f34d
PLC_PRG --|1|--> R_TRIG
PLC_PRG --|5|--> real_mul


Totals:
1 calls to INTEGRAL_INIT <=> 212608709946d8bc4c358a104cfd97642d83ed5f81c5efba06f1a2de12cfcdbd
7 calls to ulong_to_real
2 calls to real_eq
12 calls to real_div
4 calls to PID_FIXCYCLE_INIT <=> 6272173c4637da0257fb72e6d6b1ae85f9844ae88eb7fb48eef44436e3b7af7f
1 calls to SysDebugHandler
12 calls to real_add
6 calls to real_gt
4 calls to INTEGRAL <=> 56a027c1dd7b107117764a3bc92c523095cb12a58dbddc5da910eda74e7619f3
13 calls to real_sub
3 calls to real_lt
1 calls to real_to_long
2 calls to R_TRIGinit
1 calls to SUB_2 <=> 8aabf99b2d021bd68a716d40545777fd22c469d97a107c6c7e1face2213c21a1
4 calls to real_ne
1 calls to DERIVATIVE_INIT <=> c859bbdedfbfc39a1a21860da25b647049137fd6f83a79dd7dd324f5c69d7d7b
2 calls to PID_FIXCYCLE <=> 113d173c5712ef3fa03065f44b109516cabc7af102bf25f590c43983def3f34d
2 calls to DERIVATIVE <=> bf1d54d93d1bebfff20c45eb889b81c8ffe7ade0e10cf3d3a9419a7a4f7b3116
1 calls to float_to_dword
1 calls to R_TRIG
18 calls to real_mul
reversing@icsref:$
```

Fig. 4. ICSREF pidargs of in house PRG.



graph_TE	11/18/2025 11:13 AM	Chrome HTML Document	25 KB
sub_2e64.disasm	11/18/2025 11:13 AM	DISASM File	12 KB
sub_11c8.disasm	11/18/2025 11:13 AM	DISASM File	16 KB
sub_15d4.disasm	11/18/2025 11:13 AM	DISASM File	2 KB
sub_26dc.disasm	11/18/2025 11:13 AM	DISASM File	30 KB
sub_50.disasm	11/18/2025 11:13 AM	DISASM File	58 KB
sub_1150.disasm	11/18/2025 11:13 AM	DISASM File	2 KB
sub_1624.disasm	11/18/2025 11:13 AM	DISASM File	57 KB
sub_2544.disasm	11/18/2025 11:13 AM	DISASM File	6 KB
sub_e20.disasm	11/18/2025 11:13 AM	DISASM File	1 KB
sub_e48.disasm	11/18/2025 11:13 AM	DISASM File	1 KB
sub_e60.disasm	11/18/2025 11:13 AM	DISASM File	1 KB
sub_e70.disasm	11/18/2025 11:13 AM	DISASM File	1 KB
sub_eb0.disasm	11/18/2025 11:13 AM	DISASM File	10 KB
TE.analytics	11/18/2025 11:14 AM	ANALYTICS File	4 KB
TE_init_analysis.dat	11/18/2025 11:13 AM	DAT File	291 KB

Fig. 5. Disassembled files generated by ICSREF of in house PRG.

our demonstration capabilities. In the paper they provided a case study using it to generate and inject a payload that they created from reverse engineering the binaries and altering a key variable. The code given on GitHub has a limited number of functions to use to get the CFGs of binaries, but none to recreate binaries as they demonstrated.

For our demonstration of the use of the ICSREF we had access to all of the sample binaries that they provided, but were limited to reverse engineering these libraries to get header information, subroutines, and the symbol table that contains information about variable names, function names, and their memory addresses. This then allowed us to generate the CFG using the data that was collected such as the functions and the linking between them. This is the extent of what we were able to create with the framework, however this was still extremely useful information created from just the binaries. This can be used for further research into the security of these PLC binaries as well as used for malicious purposes.

VIII. Conclusion

Although IT and OT environments have expanded over the years, many Industrial Control Systems have PLCs that have been in place for many years lacking modern protections against cybersecurity attacks. Whether it may be preventing attacks or keeping them from going undetected, the security of these PLCs are more important than ever. The ICSREF framework provides a useful advancement that allows for the automated analysis of PLC binaries. This speeds up the process of reverse engineering the binaries to recover program structures, key variables, and I/O interactions. In the past this has been more difficult due to different proprietary frameworks, compilers, and development environments, but is solved by knowledge databases collected by the authors. The ICSREF framework aims to accelerate the analysis of binaries for detection of malicious code and improve the security of how they are programmed.

Our analysis of the ISCREF framework has given a better understanding how the reverse engineering of binaries can essentially recreate the original program code indicating vital components that may influence the performance of PLCs. We showed that with a framework such as this, attackers could determine variables and I/O operations and generate altered binaries that discretely change how a PLCs functions such as as operating points. We were able to go as far as demonstrating the recovery of functions and the linking between them by generating our own Control Flow Graphs. However, we were limited by the nondisclosure of some of the more important intellectual property created by the authors such as recreating modified binaries after analyzing. Through this study, we reinforced how essential tools such as the ICSREF framework are and will be in maintaining secure industrial control environments.

References

[1] A. Keliris, H. Salehghaffari, M. Maniatakos, P. Krishnamurthy, and F. Khorrami, "ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, Dallas, TX, USA, 2017, pp. 2283–2298.

[2] A. Keliris, H. Salehghaffari, B. Cairl, P. Krishnamurthy, M. Maniatakos and F. Khorrami, "Machine learning-based defense against process-aware attacks on Industrial Control Systems," 2016 IEEE International Test Conference (ITC), Fort Worth, TX, USA, 2016, pp. 1–10, doi: 10.1109/TEST.2016.7805855.

[3] D. Bhamare, M. Zolanvari, A. Erbad, R. Jain, K. Khan, and N. Meskin, "Cybersecurity for industrial control systems: A survey," *Computers & Security*, vol. 89, 2020, 101677, ISSN 0167-4048.

[4] J. Lee, T. Avgerinos and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," 2011 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 2011, pp. 1–14.

[5] D. Kushner, "The real story of Stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, March 2013, doi: 10.1109/MSPEC.2013.6471059.