Abstract—The security of Industrial Control Systems (ICS) has been attracting increased attention over the past years, following the discovery of real threats targeting industrial environments. Despite this attention, automation of the reverse engineering process of ICS binaries for programmable logic controllers remains an open problem, mainly due to the use of proprietary compilers by ICS vendors.（ICS供应商使用了专有的编译器使得可编程逻辑控制的ICS二进制程序的你想过程自动化仍然是一个公开的问题） Such automation could be a double-edged sword; on the one hand it could accelerate digital forensic investigations and incident response actions, while on the other hand it could enable dynamic generation of malicious ICS payloads（这也使得能够动态生成恶意的ICS payload）. In this work, we propose a structured methodology that automates the reverse engineering process for ICS binaries taking into account their unique domain-speciﬁc characteristics（结构化的方法考虑ICS特定领域的特征，自动的逆向ICS二进制文件）. We apply this methodology to develop the modular Industrial ControlSystemsReverseEngineeringFramework(ICSREF),and instantiate ICSREF modules for reversing binaries compiled with CODESYS（作者逆向了CODESYS）, a widely used software stack and compiler for PLCs. To evaluate our framework we create a database of samples by collecting real PLC binaries from public code repositories, as well as developing binaries in-house. Our results demonstrate that ICSREF can successfully handle diverse PLC binaries from varied industry sectors, irrespective of the programming language used. Furthermore, we deploy ICSREF on a commercial smartphone which orchestrates and launches a completely automated processaware attack against a chemical process testbed. This example of dynamic payload generation showcases how ICSREF can enable sophisticated attacks without any prior knowledge。

I. INTRODUCTION

Industrial Control Systems (ICS) are systems used to control, monitor, and interconnect physical processes in industrial settings. A wide variety of sectors rely heavily on ICS for their operation; examples include the oil and gas industry, food processing, electric power systems, and water treatment and desalination facilities. As evident from these examples, ICS often control national critical infrastructure. Any disruption to their operation can have far-reaching consequences, ranging from severe ﬁnancial losses to environmental disasters, and even loss of life [44].（工控安全影响大）。

Over the past years, threats originating from the cyber domain have become potential sources of disruption for ICS, following the discovery of real ICS cyberattacks[36].The most prominent example is Stuxnet, an attack against a uranium enrichment facility in Iran in 2010 [13]. Other examples include two attacks against the Ukrainian power grid, which led to partial blackouts in 2015 and 2016 [28], [10], and an attack against petrochemical safety systems in Saudi Arabia [22]. From a ﬁnancial standpoint, the annual cumulative losses of cyber incidents for large ICS companies can be in the range of $500,000 USD [23]. Correspondingly, the global ICS security market is expected to grow from $10.24 billion USD in 2017 to $13.88 billion USD by 2022 [29].（影响大）

The main enabler of this new class of attacks is the ongoing convergence between Operational Technology (OT) and Information Technology (IT). To reduce costs, Commercial-OffThe-Shelf (COTS) components are increasingly being used in ICS hardware and software. At the same time, ICS components are being ubiquitously interconnected for increased situational awareness, better control, and enhanced efﬁciency. Despite the many beneﬁts of the IT/OT convergence, an unwanted side-effect is the exposure of ICS environments to the same cybersecurity risks that plague the IT domain. As recent ICS attacks demonstrate, attackers are taking advantage of the expanded ICS threat landscape during this transitional period, whereas defenders appear to be lagging behind.（ICS广泛连接使得更方便但是更容易受威胁）

ICS control physical processes through Programmable Logic Controllers (PLCs). Process engineers program these controllers by developing code which is compiled to a binary that controls the target PLC. An important objective, both for actors who wish to protect, as well as actors who wish to attack ICS processes, is to automatically reverse engineer PLC binaries（是关键）. On the one hand, such capability could speed up actions after an ICS cyberattack. Digital Forensics and Incident Response (DFIR) teams could leverage automated reverse engineering to understand attack objectives of PLC malware, and timely deploy countermeasures. On the other hand, ICS malware designers could leverage the same capabilities for dynamic payloads that do not require communication with a Command and Control (C2) center, enabling sophisticated attacks even against targets that reside in air-gapped networks（一个漏洞能打一堆）.

To date, to the best of our knowledge, the problem of automatically reverse engineering ICS PLC binaries remains an open problem [30], [26]. ICS binary reverse engineering remains a tedious manual procedure carried out by domain expert reverse engineers that are extensively familiar with the systems under study. Recent academic literature on ICS security also highlights that automated reverse engineering is an important, unsolved problem; authors either make over simplifying assumptions regarding the binaries of interest, or assume that information for the physical process is obtained utilizing other means (e.g., espionage) [14], [2], [31], [15], [49]. In addition, the tools, frameworks, and techniques for reverse engineering binaries (e.g., the Interactive Disassembler — IDA Pro) in the IT world do not directly translate to the OT world [34]（IT逆向工具不能被OT使用）. One of the reasons is that ICS vendors typically employ proprietary and/or not well-documented compilers for generating binaries for their target PLCs（不能被逆向的原因是使用专有或小众的编译器编译PLC二进制文件）; Siemens PLCs are programmed with the SIMATIC STEP 7, Allen-Bradley PLCs with the Studio 5000 Logix Designer and the majority of other ICS vendors employ the CODESYS framework.

Identifying the gap in methodological approaches, as well as the lack of frameworks and tools speciﬁcally tailored to PLC, in this paper we propose a methodology that considers the unique characteristics of PLC binaries, and introduce the Industrial Control Systems Reverse Engineering Framework (ICSREF). ICSREF automates the reverse engineering process for ICS binaries and can provide information on the physical characteristics of a system captured in the ICS binaries controlling it, without any prior knowledge of the system.

Automated reverse engineering of PLC binaries can enable a broad range of studies. On the one hand, the semantic-rich information stemming from ICSREF analyses can be employed for enhancing the security of ICS（ICSREF分析丰富的信息能够增强ICS安全性）. A defensive use case that highlights the need for fast, automated reverse engineering of PLC binaries is PLC malware analysis. For example, immediately after the discovery of Stuxnet, there was a pressing need for reverse engineering the malicious STEP7 payload to understand its impact on the nuclear plant it targeted. The manual reverse engineering process carried out was arduous and time consuming, delaying forensic investigations. Falliere, who reverse engineered Stuxnet, subsequently created the JEB decompiler for S7 PLCs, which automates the process and expedites analysis of Siemens S7 PLC binaries [37]. Similarly, ICSREF can be employed to analyze PLC malware that target CODESYS-enabled PLCs, assisting DFIR professionals to quickly determine the functionality and objectives of malicious PLC binaries. When analyzing PLC malware, ICSREF can also assist in PLC code authorship attribution by identifying malicious actors through ﬁngerprinting the code they utilize. In the same way, ICSREF function and code snippet ﬁngerprinting can help legitimate actors prove IP copyright infringement by their competitors. ICSREF analyses can additionally aid source code recovery and binary code reuse in the case of legacy PLCs for which the original source code is not available.（在防御方的作用）

On the other hand, ICSREF highlights the signiﬁcance of adopting proper, proven security practices in ICS environments. Automated reverse engineering can, for example, enable dynamic process-aware payload generation, lowering the requirements for malicious actors, and allowing sophisticated attacks against air-gapped systems without requiring prior knowledge.（在攻击方的作用）

Our contributions can be summarized as follows:

We propose a structured methodology for reverse engineering generic PLC binaries, which captures their intricacies and unique characteristics.（结构化方法逆向PLC二进制文件，能够捕获二进制文件的复杂性和独特特点）

We develop ICSREF, a reverse engineering framework for PLC binaries, which automatically analyzes binaries created with the CODESYS platform and fully reconstructs their Control Flow Graph (CFG).（针对CODESYS架构逆向，并完全重建CFG图）

We collect and consolidate PLC source code and binaries from public code repositories, creating a database of samples for further studies（收集PLC源代码）.

We carry out an end-to-end case study of dynamic payload generation and attack deployment against a chemical process. A commercial smartphone equipped with ICSREF automatically generates and deploys the attack, without any prior knowledge of the process.（动态生成payload，用智能手机自动攻击）

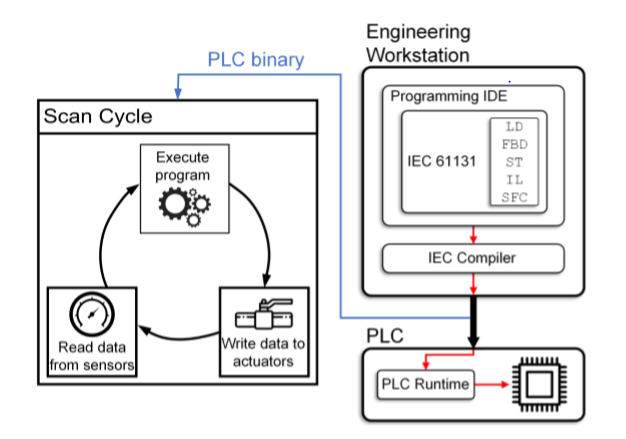
The remainder of the paper is organized as follows: In Section II we provide background information on ICS and PLC binaries, and identify their unique characteristics compared to conventional binaries（二为背景知识，ICS的独特之处）. We propose a structured reverse engineering methodology speciﬁcally tailored to PLC binaries in Section III（三为介绍逆向引擎的实现）. In Section IV we introduce the ICSREF framework and present its technical details（四为框架的技术细节）. Section V presents our techniques and results for validating ICSREF correctness, including the creation of a database of real binaries collected from public code repositories, and a performance evaluation of our framework.（五为评估） We demonstrate an end-to-end case study of automated attack formulation and deployment from a commercial smartphone enabled by ICSREF in Section VI.（六为部署） We compare ICSREF with related work in Section VII（七为比较）, and conclude the paper in Section VIII.（八为总结）

## II. PRELIMINARIES

In this section we provide background information on ICS environments and outline the unique characteristics of PLC binaries1 compared to conventional binaries.

1. Industrial Control Systems

ICS is a broad term that encompasses various types of conﬁgurations for controlling and monitoring industrial processes. Such conﬁgurations include Supervisory Control And Data Acquisition (SCADA) systems for geographically dispersed systems, Decentralized Control Systems (DCS) for large industrial processes with autonomous controllers, and Process Control Systems (PCS) for small industrial settings. ICS can be abstracted as a set of control algorithms that operate on sets of measurement values obtained from the physical environment via sensors. The results of these control algorithms are in turn used to regulate the controlled physical processes via actuators [35]. The process of sensing the environment, calculating control signals and updating outputs on actuators is also known as the scan cycle (left side of Fig. 1), and is usually performed continuously and in a realtime fashion by PLCs [44]. PLCs are real-time embedded systems encased in ruggedized packages to withstand the harsh industrial environments they are deployed into. They typically include one or more microprocessors, volatile and non volatile memories, analog and digital I/O subsystems, and networking capabilities [34].（介绍ICS）



Sensros 传感器 actuators 执行机构

1. Software development of PLC programs

The International Electrotechnical Commission (IEC) deﬁnes industry standards for PLCs in IEC 61131. More specifically, the third part of the standard, IEC 61131-3, concerns software architecture and programming of PLCs, including programming languages, datatypes, variable attributes, etc. [20]. IEC 61131-3 describes the following graphical and textual programming languages for PLCs:

• Ladder Diagram (LD), graphical

• Structured Text (ST), textual

• Function Block Diagram (FBD), graphical

• Sequential Function Chart (SFC), graphical

• Instruction List (IL), textual (deprecated)

（IEC对PLC标准进行一些规定）

The PLC software development process is depicted in Fig. 1. Process engineers develop PLC logic at engineering workstations. The engineering workstations are equipped with vendor-provided IEC 61131-3-compliant Integrated Development Environments (IDEs) and compilers for the speciﬁc PLC models used in the plant（供应商提供IDE）. PLC logic for controlling the physical process is developed using one or more of the IEC 61131-3 languages listed above and is then compiled using an IEC compiler. The binary is transferred to the PLC, a procedure called program download in ICS terminology（下载：二进制文件从IDE传输到PLC）, where a PLC runtime (a process executed by the PLC’s operating system or ﬁrmware（运行：PLC操作系统或固件执行过程）) handles the binary loading and execution, enforcing the real-time requirements and enabling debugging and monitoring of the PLC binary execution（实时执行并调试和监视PLC二进制文件的执行）. During normal operation the binary is loaded to the PLC’s fast, volatile memory and executes from it（加载到快速易失内存中执行）. To ensure fast recovery in the event of an outage, the PLC binary is also stored in non-volatile memory（为确保故障恢复，PLC二进制文件也存储在非易失性存储器中）, usually accompanied by retain/persistent variables that conserve information regarding the state of the system（通常伴随保存持久变量以保存系统状态信息）. From there, the PLC binary and the system execution state can be automatically loaded upon PLC startup and resume execution even after uncontrolled halts of the program.（PLC运行信息保留两部分，一部分是易失性，主要讲了非易失性存储一些运行时的全局变量和运行状态用于启动和暂停后的恢复）

1. PLC binaries vs. conventional binaries（PLC和传统二进制的区别 ）

Analyzing PLC binaries requires understanding their unique domain-speciﬁc characteristics that differentiate them from binaries targeting conventional workloads. Here, we highlight the main differences between the two, and discuss how these differences can facilitate or impede analyses.

**Execution model**: Besides differences in programming paradigms, execution models between conventional and PLC binaries are also different. Non-PLC languages, and by extension their compiled binaries, usually follow sequential execution of units of work（非PLC遵循工作单元运行，就像c成都分号） (e.g., ;-delimited statements for the C programming language [25]). On the contrary, the execution model of PLC binaries is dictated by the scan cycle, inﬁnitely executing its three comprising steps. This can hinder dynamic analyses on entire PLC binaries due to their inﬁnite execution nature, requiring that appropriate code sections are carved out for individual dynamic analyses.（PLC由扫描周期决定，无限重复三个步骤，需要对各个动态分析指令指定相应的代码段）

**I/O operations:** Although the majority of conventional binaries rely on I/O operations to get input variables and produce corresponding outputs, the importance of I/O operations for PLC binaries is signiﬁcantly higher. I/O operations in PLC binaries are a critical and necessary part of their functionality, occupying two thirds of the scan cycle. PLCs are by design devices for interacting and controlling the physical world, something that is achieved through sensor and actuator signals via the PLC I/Os. It is thus necessary to identify the mechanisms with which a PLC binary interacts and orchestrates read and write operations to these I/Os.（I/O对PLC很重要，需要知道详细细节）

**File format:** Binaries compiled for major Operating Systems (OSs) usually follow well documented formats, for example the Executable and Linkable Format (ELF) for Linux and the Portable Executable (PE) format for Windows. These widely used formats are handled by loaders of their respective OSs. On the contrary, loading of PLC binaries is typically handled by proprietary loaders (e.g., the CODESYS runtime [16]), and the ﬁle format of PLC binaries is custom and unknown. This impedes the analysis of PLC binaries, as these custom proprietary formats need to be ﬁrst reverse engineered to allow further exploration.（PLC自定的可执行文件格式，需要知道详细细节）

**Optimization**: Due to requirements for provable enforcement of real-time execution deadlines and assurances dictated by various standards, compilers for PLC binaries typically only make very conservative optimizations（为满足各种要求PLC通常只做很保守的优化）, if any [21]. On the contrary, conventional binaries that do not control critical environments typically employ several aggressive compiler optimization techniques [3]. While this may lead to larger and less efﬁcient PLC executables, it can also facilitate reverse engineering as they will be functionally simpler.（很少的优化有利于逆向）

In general, these differences can be attributed to the mission-speciﬁc nature of PLCs, as well as the long history of industrial automation hardware and software that did not always follow the progression of general purpose computers [17]. Any reverse engineering efforts should be informed by these deviations from conventional binaries.

## III. PROPOSED METHODOLOGY FOR REVERSE ENGINEERING PLC BINARIES

In this section, we propose a structured overarching methodology that is speciﬁcally tailored for reverse engineering PLC binaries. To date, several of the binary analysis approaches and techniques proposed for arbitrary binaries

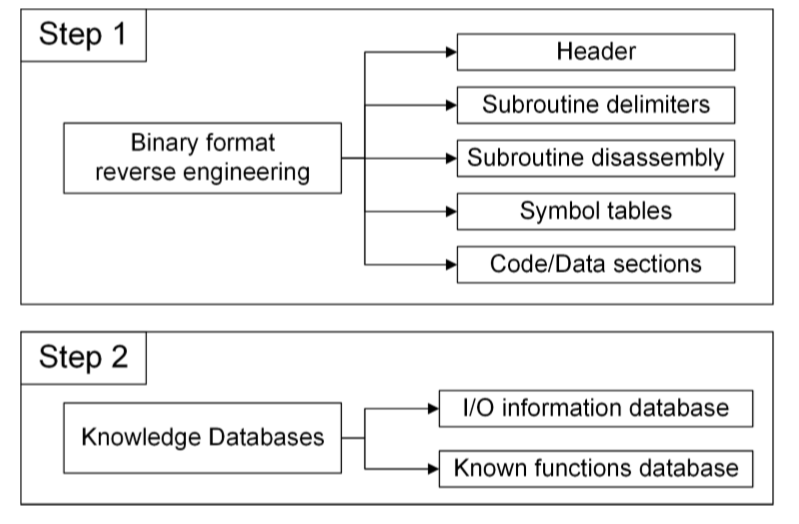


Fig 2 Steps of platform-speciﬁc phase

Subroutine delimiters 子例程 分隔符。

Disassembly 拆卸

(e.g., function identiﬁcation [4] and type inference（推理） [6]) can be leveraged for individual subtasks of PLC binary analysis. However, to the best of our knowledge, a systematic, structured methodology that covers the entire reverse engineering process for PLC binaries from start to ﬁnish is lacking. With our proposed methodology we aim to address this gap, by identifying the necessary steps and required outcomes.(现有工具可以用于单个子任务的plc逆向，但是一个系统的结构化的方法是缺乏的)

When reverse engineering PLC binaries, both the similar-ities and differences between conventional and PLC binaries should inform the possible approaches, techniques, and tools. Taking into account their unique properties and their nature, we propose a structured methodology for enabling typical automated reverse engineering tasks and objectives, consisting of two phases:

1. A platform-specific phase, carried out once for each platform (e.g., STEP7, Studio 5000, or CODESYS).During this phase, we extract general information that characterizes and applies to all the binaries generated by that platform.(针对特定平台：取出该平台生成二进制文件的所有特征和二进制文件的一般信息)
2. An automated binary analysis phase, carried out for every binary. During the second phase, one must automate the extraction of information from a PLC binary, leveraging the platform-specific results.(二进制自动分析阶段：利用1的结果自动从plc 二进制文件中提取信息)
3. Platform-specific phase

This phase aims to extract information concerning the specifics of a platform. We break down this phase into two steps: 1) reverse engineering and understanding the general format of PLC binaries compiled with that platform, and(逆向引擎理解文件格式) 2)creating knowledge databases（包含有助于二进制分析的常规信息）, that contain general information that can facilitate automated binary analyses. A schematic representation of the platform-specific phase is presented in Fig. 2. In general, this phase incurs a one-time cost per platform, and requires a minimal instantiation of the platform under analysis (i.e., a PLC device that uses this platform, and its corresponding IDE to generate PLC binaries).

1. Binary format reverse engineering:

The majority of ICS platforms employ proprietary, undocumented binary formats in their PLC binaries. Thus, the ﬁrst step consists of understanding these formats and extracting relevant information. Similar to widely used executable binary formats, PLC binaries may include one or more headers, code and data sections, symbol tables for dynamically linked code, etc.（plc的格式和普通二进制文件相似）There are no deﬁnitive methods for reverse engineering unknown ﬁle formats, however reverse engineering techniques from domains such as embedded systems ﬁrmware reverse engineering and ﬁle format analysis can assist in this step [8], [41], [48], [11]. In addition to the proprietary nature of PLC binary formats, another impediment for this step is the execution model of PLC binaries.（难点1.专有格式2.PLC二进制文件的执行模型） Because of the inﬁnite nature of the scan cycle, dynamic analyses cannot be performed on the entire PLC binary, but only on appropriately carved out instruction sequences（无限的循环执行使得不能对整个二进制文件执行动态分析，而是对指令适当的划分）. Nevertheless, lack of optimizations in PLC binaries may be beneﬁcial for this step, as generalizations can be more easily made due to the immutable compilation results.（PLC缺少优化有利）

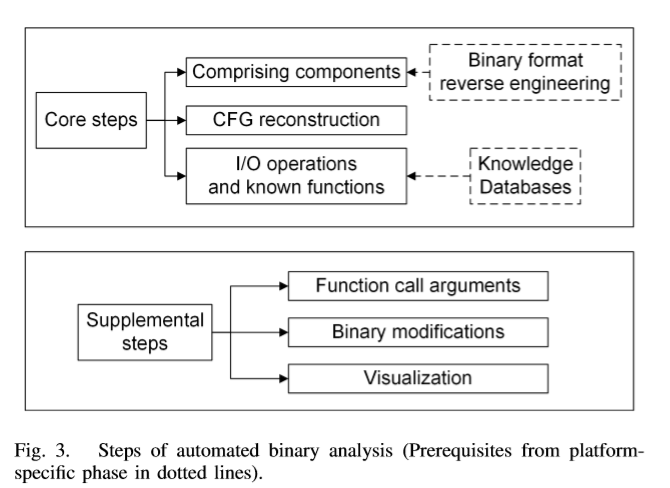
In general, the results of the binary format reverse engineering step should include information on header contents, how subroutines are delimited, extraction of the subroutines and their disassembly listings, identiﬁcation of symbol tables and dynamically linked functions, as well as information on code and data sections.（二进制格式逆向工程步骤的结果应包含一下内容：标头内容，如何分割子例程，提取子例程及其反汇编列表，标识符号表和动态链接的函数以及有关代码和数据的信息）

2) Knowledge Databases: For the second step, we identify the need for creating two knowledge databases containing general information that enables and accelerates reverse engineering tasks. The ﬁrst should include information pertaining to I/O operations performed in the binary, given their signiﬁcance for PLCs. This I/O information database should include information on how a binary reads/writes from/to physical I/Os. In the common case of memory-mapped peripherals, the corresponding addresses for these I/O peripherals should be identiﬁed and included in the database.（知识数据库包含 I/O的信息）（I/O设备对应的地址）

The second database should contain signatures of known library functions and code snippets（包含二进制文件的代码片段的签名）. These can ﬁngerprint known subroutines in arbitrary binaries, reducing the manual effort required by a reverse engineer（任意二进制文件中识别指纹已知的子程序）. This known functions database should contain information pertaining to standard functions and libraries that can be statically linked in the binary（包含标准函数和库的信息）. Similar to typical programming paradigms, IEC 61131-3 programmers can import and utilize library Functions/Function Blocks (F/FBs) in conjunction with their own F/FBs. Typical examples include F/FBs that handle network communication (e.g., MODBUS/SMTP stacks), common control algorithms (e.g., PID), and timing functions (e.g., triggers, timers). Identifying and ﬁngerprinting these functions can speed up the analysis of a given binary, as the reverse engineer will not have to spend time understanding already known functions. A similar approach is employed by the commercial IDA Pro disassembler with its F.L.I.R.T. technology [18]. The corresponding knowledge database should employ a signature scheme that ensures low false positive and false negative rates when ﬁngerprinting subroutines.（知识数据库需要包含已知函数的签名，识别出已知函数）

1. Automated binary analysis

The second phase automates the reverse engineering process of arbitrary binaries. It assumes that binaries are developed using a known platform analyzed during the previous phase. We identify a set of three necessary core steps for performing automated analyses, and propose supplemental steps that facilitate common reverse engineering tasks. Fig. 3 summarizes the requirements and components of this phase.（图三是本阶段的需要和组成的总结）



Reconstruction 重建

Supplemental 补充

Modification 修改

Visualization 可视化

Prerequisite 先决条件

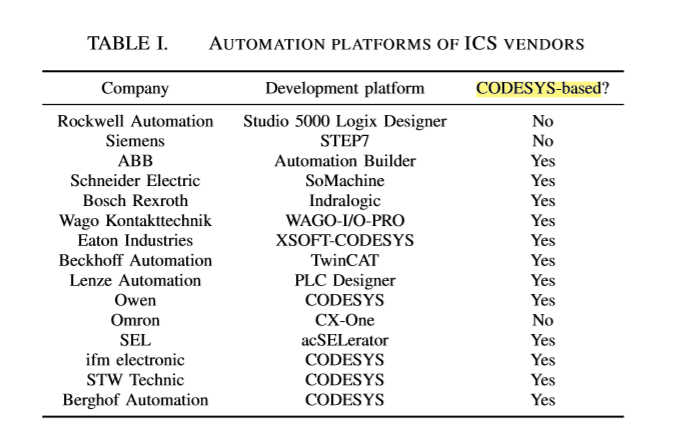
The ﬁrst core step should include the dissection of the binary to its comprising components, leveraging the binary ﬁle format information from the previous phase（利用前一阶段的格式信息将二进制文件分解到其组成的组件）. In this step, all subroutines must be delimited and disassembled, and code/data section and symbol tables describing dynamically linked functions identiﬁed（需要分割和反汇编子过程，并识别描述动态链接函数的代码段数据段和符号表）. The second core step concerns the reconstruction of an as complete and as sound as possible CFG, by ﬁnding and resolving the targets of branches. We use the deﬁnition of recovered CFG soundness and completeness from [39]. Finally, the third core step should identify instructions resulting to I/O operations and ﬁngerprint known functions, utilizing the knowledge databases.

Supplemental steps can augment the resulting analysis and allow further automation. For example, a supplemental step concerns the automated extraction of arguments passed to function calls, using dynamic binary analysis techniques and symbolic execution [39]（找到函数的参数）. This step can be leveraged to extract semantic information regarding the physical environment that a PLC binary is controlling, as this is captured by the binary（参数能够用来提取关于PLC二进制文件所控制物理环境的语义信息）. It can also assist in recovering lost source code, re-engineering and reuse of code and intellectual property audits [12], [47]. Another supplemental step considers modiﬁcation of binaries. This enables dynamic payload generation [30], or injection of host-based defenses [7] （修改二进制文件，用于payload自动生成，或者注入基于主机的防御）. Binary modiﬁcation should also take into account any ﬁle integrity mechanisms, such as CRC checks [43]（需要考虑文件完整性）. A further step should provide intuitive representation of results, including CFG visualization. In our implementation of ICSREF we implement all these supplemental steps, as described in the following section.

The structured methodology we describe in this section is general, as it is designed taking into account vendor-independent characteristics of PLC binaries. As such, it can be followed for reverse engineering of PLC binaries irrespective of development platform.（这种设计方法是跨平台的）

## IV. THE ICSREF FRAMEWORK

In this section we present the technical details of the Industrial Control Systems Reverse Engineering Framework (ICSREF), our novel modular automated reverse engineering framework. Applying our proposed methodology, we instantiate ICSREF modules that can carry out automated analyses for CODESYS-compiled binaries.（对三中的设计进行实例化做出了ICSREF，可以分析CODESYS架构的二进制文件）



Vendors 供应商

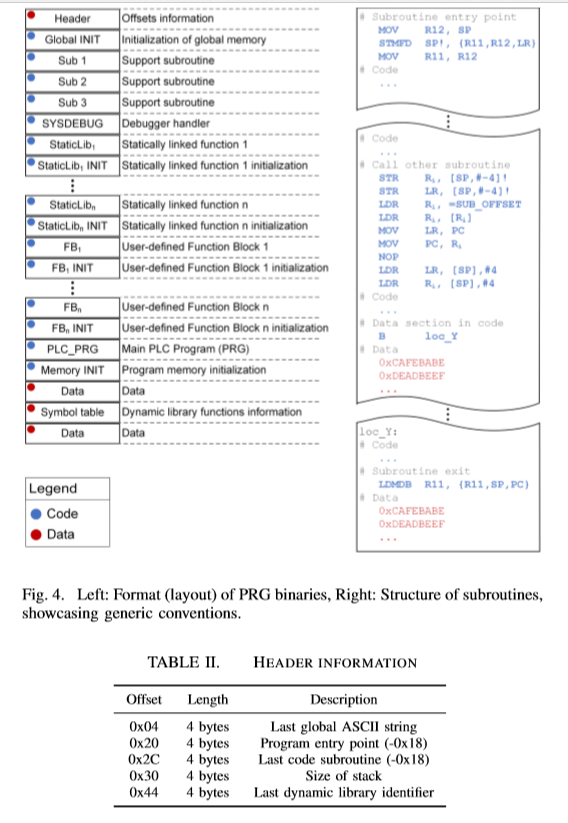
CODESYS is a hardware-independent IEC 61131 platform for industrial automation applications. We focus on CODESYS because it is widely used; more than 250 manufacturers employ it in their products, including multinational corporations with substantial ICS market share [16]. The CODESYS Device Directory lists 349 products or family of products that are programmed with CODESYS and are currently supported [1]. The actual number of CODESYS-enabled devices is much larger as several products no longer appear in the directory but are still deployed. Table I lists software development platforms used by major automation corporations, demonstrating the widespread adoption of CODESYS. Due to the diverse categorization of industries involved in automation (plant automation, transportation, energy, process automation, building automation) and the sparsity of published data regarding market share of these companies, we do not have deﬁnitive data regarding CODESYS market share. However, from our domain expertise and discussions with experts, our conservative estimate is that at least 20% of PLCs worldwide employ CODESYS.（CODESYS用的十分广泛，至少20%）

1. Platform-speciﬁc phase: CODESYS

For the needs of the ﬁrst phase, we begin by reverse engineering known binaries towards understanding the binary format of CODESYS PLC binaries.（先知道格式） Subsequently, we leverage our ﬁndings to automatically create the two knowledge databases containing PLC I/O memory maps, and known library function signatures（创建知识数据库包含I/O内存地图和已知库函数签名）. We carry out these studies utilizing a WAGO 750-881. The 750-881 employs an ARM microprocessor and uses CODESYS v2.3. Regardless, we stress that the results presented hereafter are not limited to binaries targeting WAGO PLCs, but are applicable to any binary compiled with CODESYS, irrespective of manufacturer.

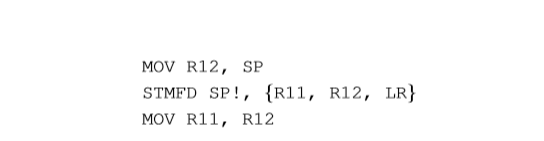
1. Binary format reverse engineering:

In understanding the format used in CODESYS PLC binaries, we start by developing a small number of programs that perform simple tasks. For this we utilize the CODESYS IDE, and write programs in all available IEC 61131-3 languages（在所有的可用的。。。语言）. We compile the programs for the ARM architecture employed by the WAGO 750-881（应该是一个编译器）, obtaining the corresponding PLC binaries (the ﬁle extension for WAGO CODESYS binaries is PRG). We focus on ARM binaries because of the proliferation of ARM processors in embedded systems deployed in ICS [33]. Preliminary investigations of binaries compiled for other architectures indicate that the ﬁndings we describe in this section are representative of overarching CODESYS compiler mechanics, and small modiﬁcations sufﬁce for extending ICSREF to other hardware architectures（这里知识arm想要其他架构还需要一些简单修改）. Subsequently, we examine and reverse engineer these in-house generated PRG binaries. We disassemble the binaries and spend a considerable amount of time comparing their disassembly listings towards understanding the undocumented CODESYS binary format and the various compiler conventions.（反汇编了PRG并分析编译器约定） Fig. 4 summarizes our ﬁndings regarding binary format and code subroutines structure.



Header: The ﬁrst 80 bytes of a PRG binary constitute a header that contains general information. Table II outlines the most important information. For example, the value obtained by adding 0x18 to the 4 byte value at offset 0x20 within the header provides the program’s entry point. The resulting value is the location of the Memory INIT subroutine. We note that it is sufﬁcient to only reverse engineer part of the header for enabling automated analyses.

Subroutine delimiters: We pinpoint the entry and exit instruction sequences (i.e., subroutine prologue and epilogue) that can delimit subroutines included in a binary. At the beginning of subroutines, CODESYS-compiled binaries use the following instructions:（通过调用约定找到函数的开始和结束）（这个可以用来识别函数的开头和结尾！！！）



For exiting a subroutine the following instruction is used:

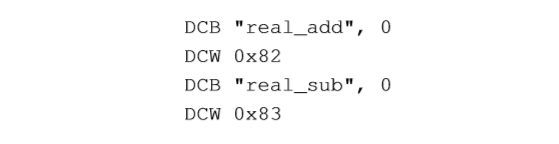


An advantage stemming from the lack of optimizations in PLC binaries, is that these delimiters are unique and universal for all ARM CODESYS-compiled binaries（二进制文件缺少优化会使得这些分隔标志对所有arm编译的二进制文件都是唯一和通用的）. We subsequently disassemble each subroutine and study its instructions in depth to derive its functionality. Through this approach, we reverse the entire PRG binary format, as shown in Fig. 4.（分析得到了其他的格式）

Initialization of global variables and generic subroutines: The ﬁrst subroutine (Global INIT) starts at offset 0x50, following the header. This subroutine sets constants, variables, and initializes functions deﬁned in the VAR\_GLOBAL（Global INIT完成的任务） section of an IEC 61131-3 program. It is common practice for PLC programmers to use this section for deﬁning program-wide constants regarding the physical environment under control（用来定义全局变量） (e.g., scaling factors, PID gains, timing constants). After the global initialization subroutine,

Statically linked libraries and user-deﬁned Function Blocks: The debugger handler is followed by subroutines for imported library F/FBs. Each statically linked F/FB consists of two subroutines: one that performs its main functionality (StaticLibi), and one that initializes its local memory (StaticLibi INIT). User-deﬁned F/FBs that correspond to code directly developed by a PLC programmer are placed after the library F/FBs in a similar way: ﬁrst a subroutine that performs their main functionality (FBi), followed by its initialization subroutine (FBi INIT). We also identify the penultimate subroutine to be the main function, a userdeﬁned FB named PLC\_PRG in CODESYS. This subroutine is mandatory and serves as the starting point of the scan cycle.（定义倒数第二个子例程标识为主函数，名为PLC\_PRG,作为扫描周期的起点）

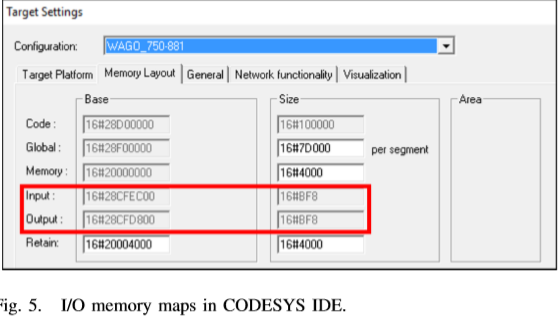
Symbol table: Simple standard functions, such as mathematical operations on REAL-typed variables, are dynamically linked in CODESYS binaries. Information about these functions is included in a symbol table, that is located after the last code subroutine. The symbol table contains sets of null-terminated string identiﬁers, followed by two bytes of data, as follows:（以null结尾的字符串，后面跟两个字节数据）



These two data bytes are used by the runtime to calculate the jump offset required for calling the corresponding function.（运行时这两个数据计算出相应函数所需跳转偏移量）

1. CODESYS Knowledge Databases:

Having understood the ﬁle format and taking into account the unique characteristics of PLC binaries, we proceed to the second step with the creation of knowledge databases. To achieve this



we identify I/O operation mechanics for PRG binaries, and construct ﬁngerprints for identifying known functions.（为PRG文件确定I/O操作并构造已知函数的指纹）

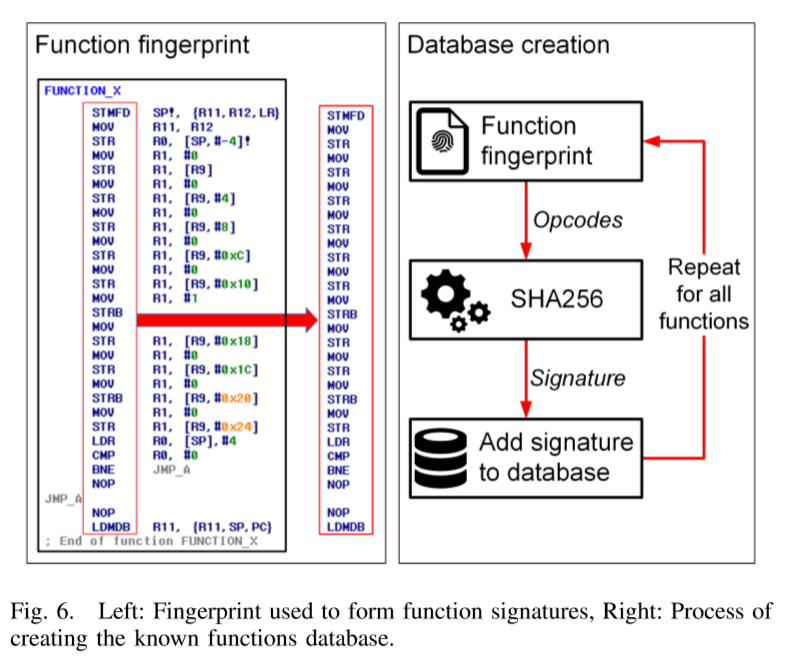
I/O database: To communicate with the environment, the physical I/O modules in CODESYS-enabled PLC devices are memory mapped to speciﬁc addresses within the memory space of a PLC（物理的I/O模块是PLC的对应内存）. These are visible in the CODESYS v2.3 IDE during creation of a new project, as seen inFig.5. For example, in the case of the WAGO 750-881 PLC, whenever a memory load operation in the binary reads from a memory address within the range 0x28CFEC00 - 0x28CFF7F8 it essentially queries a sensor, and whenever a memory store operation writes in the range 0x28CFD800 - 0x28CFE3F8 it updates an actuator.（0x28CFD800 - 0x28CFE3F8是I/O设备的地址空间）

Manually extracting the values for every target PLC program from the CODESYS IDE is tedious and does not scale. By inspecting the IDE installation, we ﬁnd that all architecture options for each PLC hardware are contained in target (TRG) ﬁles, residing in the installation directory of the IDE. These are data ﬁles that do not contain readable information, leading us to suspect they are encoded and/or compressed. Indeed, by further investigating the TRG ﬁle format and performing a comparative analysis of different TRG ﬁles we identify the encoding scheme used. In particular, the scheme employs exclusive disjunction (XOR) between 2048-bit blocks of the ﬁle contents and a reused 2048-bit ﬁxed sequence. We further veriﬁed that the same scheme and sequence are used for all TRG ﬁles, irrespective of vendor and the target PLC, i.e., the technique applies to non-WAGO PLCs. This allows us to parse TRG ﬁles and extract the I/O memory maps for PLC models, populating the I/O database in an automated manner. （TRG文件包含PLC的硬件选项，保存在IDE安装目录，使用2048位进行异或并且编码方式和供应商以及目标PLC无关）（TRG文件可以位PLC模型提取I/O内存映射）

Known functions database: We create ﬁngerprints for the statically linked library functions armed with our knowledge of the PLC binary format from the previous step. The signature scheme we select for uniquely identifying each subroutine consists of the SHA-256 hash digest of the concatenated sequence of its opcode mnemonics(签名方案是每个子例程的操作码助记符的SHA-256). We select opcodes and discard instruction arguments, because the arguments may contain mutable information (e.g., absolute branch targets, usage of different registers)（只使用操作码，不用参数，因为参数可能可变比如跳转，使用的寄存器）. As we veriﬁed in our experiments, the sequence of opcodes deﬁnes the functionality of the subroutine, i.e., the opcode sequence of a library subroutine is unique for each of our library F/FBs, and are not affected by the minimal compilation options offered by CODESYS.（这里能这样用应该是因为编译器很少对代码进行优化）

We populate the known functions database with signatures for all known library F/FBs efﬁciently by automating all the necessary operations. Our automated database population workﬂow generates source code that only includes a call to one F/FB, imports the code and its requirements to the CODESYS IDE, and compiles it to obtain the corresponding PRG binary. We analyze each binary to extract the signature of the F/FB it contains, and append this signature to our database. The function ﬁngerprint and the overall procedure for creating the known functions database is depicted in Fig. 6.(代码和需求导入到CODESYS IDE并编译它获得相应的PRG二进制代码，为每个二进制文件提取它包含的F/FB签名并添加到数据库)

（这里编译器优化不多，直接用这种方式存。 对于普通二进制文件是否可以这样，一个函数对应多个hash实现精确匹配，匹配hash会不会很慢）



In total, we analyze and create signatures for all 1704 F/FBs included in the 212 libraries that are available in our CODESYS IDE instantiation. The total time required for the automated process was 11 hours and 44 minutes. Although this is a time-consuming step, it is a one-time cost. After the database is created, ﬁngerprinting subroutines in any given PRG binary resolves to a fast query of the database. During our database construction we did not ﬁnd any collisions between F/FBs that perform different operations, demonstrating the suitability of using opcode sequences as identiﬁers. Inaddition, by automating this process we can easily extend the database to include F/FBs signatures from new libraries.

1. Automated binary analysis:

ICSREF We incorporate the knowledge we extract from the CODESYS platform-speciﬁc phase, as well as the automated binary analysis steps into ICSREF. ICSREF can automatically reverse engineer PRG binaries, containing modules to carry out all the steps for the second phase as they are outlined in the methodology section. Our design goals for ICSREF are:

• Accuracy: The results obtained by applying ICSREF to a PLC binary should be accurate, including full CFG reconstruction whenever possible. （准确）

• Extensibility: ICSREF design and structure should enable easy extension to new PLC targets, microarchitectures, and IEC 61131-3 development platforms. （可扩展，扩展你到其他开发平台，除了codesys）

• Usability: Usage of ICSREF should be easy, and provide an intuitive interface to the reverse engineer.（可用性，方便上手）

To achieve these design goals we follow a modular approach, where a core module handles the core steps of the second phase. Supplemental modules extend the results, providing visualization, function call argument extraction, and binary modiﬁcations.

1. Core module: This module processes arbitrary PRG binaries by carrying out the core steps of the automated binary analysis, namely binary dissection, CFG reconstruction, and identiﬁcation of I/O operations and known functions. （二进制解析，CFG重构，I/O操作数的识别）

Comprising components: To dissect a PRG binary, the core module leverages the PRG binary format information from the previous phase. First, it parses the header and extracts the information it contains. It then scans the binary searching for subroutine delimiters, and uses them to carve out all the subroutines and generate their disassembly listings using radare2. Finally, it extracts the symbol table and identiﬁes any dynamically linked functions.（解析头，扫描二进制文件搜索以查找子例程分隔符，用分隔符找到子例程用radare2生成反汇编，提取符号表识别动态链接函数）

CFG reconstruction: For every binary analysis, automated or not, it is of utmost importance to reconstruct its CFG. This includes resolving the branches that are both within each subroutine, as well as jumps from one subroutine to another（子例程自己和跳转到其他子例程的分支）. A fundamental challenge that can hinder full CFG reconstruction is resolution of indirect branch targets [39]（间接跳转是重建cfg图的难点）. Fortunately, as we explain below, PRG binaries only contain indirect jumps for calls from one subroutine to another or to a dynamically linked function, and the locations of all such indirect branch targets are completely self contained in the binary（PRG只包含一个子例程到另一个子例程或动态链接函数的间接跳转，并且跳转目标都在二进制文件里）. Essentially, we can extract and resolve all indirect branches, resulting in a fully reconstructed CFG graph. This is a byproduct of the strict requirements and standards that govern ICS platforms; lack of optimizations avoids complicated control ﬂow operations, and provable real-time deadlines drive compilers to avoid runtime resolved indirect jumps.（没有间接跳转是满足规定的副产品，一定存在）

By observing the internal structure of subroutines (see right side of Fig. 4), we identify the instructions that orchestrate indirect jumps for calls from one subroutine to another or to a dynamically linked function to be the following:（调用子例程或动态链接函数会有下面的指令）

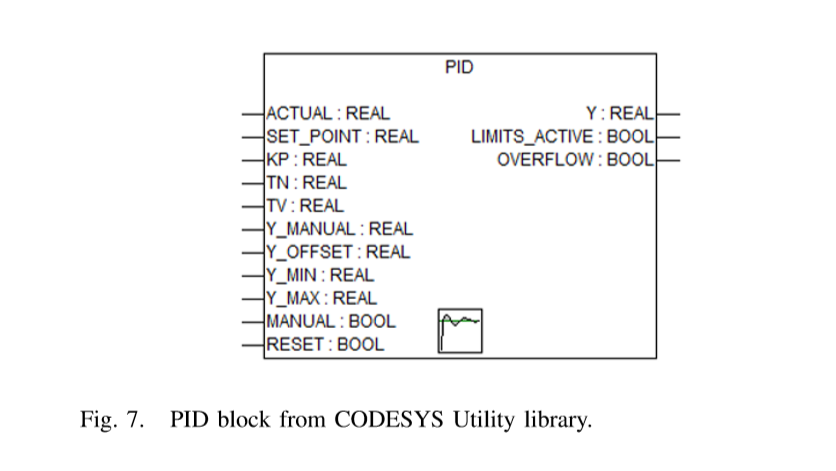


Essentially, all jump targets (SUB\_OFFSET) are contained in a call table residing in memory. （所有跳转的目标都在内存中一个调用表内）In CODESYS binaries, this call table can be fully reconstructed.（调用表能重建） For dynamically linked function targets, we identify the call table index values to be calculated from the two byte data values following the null-terminated strings within the dynamic library section（调用表索引值将从动态库section中以null结尾的字符串后面的两个字节数据值进行计算）. Speciﬁcally, the jump offset is calculated by multiplying the two byte value of the null-terminated string identiﬁer of a function by 4 and adding 8 to this result（具体来说跳转的偏移量是null结尾两字节数据值\*4+8的结果）. For in-binary subroutine targets (e.g., statically linked library F/FBs or user-deﬁned F/FBs), we identify that the call table is constructed by the CODESYS runtime after execution of the last subroutine, namely Memory INIT（call table是在codesys运行时在最后一个子例程Memory INIT运行后创建的）. The functionality of this subroutine, which is also the entry point of the binary, is twofold.（这个子例程也是二进制文件的入口点） First, it zero-initializes the memory space required by the binary. Second, it calculates the index offsets necessary for calling all the subroutines included in the binary, creating the corresponding call table.（Memory INIT：首先所需内存空间进行0初始化，计算调用所有子例程包含的所有偏移创建相应调用表）

We achieve extraction of the call table for in-binary subroutine targets by leveraging the dynamic symbolic (and concolic) execution capabilities of the angr framework（作者用angr工具提取二进制内子例程目标调用表的提取） [39], [42], [40]. Symbolic execution allows us to execute speciﬁc parts of the binary (in this case the Memory INIT subroutine) without necessarily knowing the entire state of the target system, and enables extensibility of ICSREF to other platforms.(这一步是可扩展到其他平台的) An important challenge is that the loader of angr cannot handle PRG format binaries（实际上angr不能处理PRG格式的文件）. We overcome this challenge by manually informing the analysis engine of the target architecture (in this case ARM) and setting a custom entry point for the binary at the beginning of the Memory INIT subroutine（手动解析目标架构的分析引擎，并在Memory INIT子例程开始处为二进制文件设置自定义入口）. We also force the engine to halt once it reaches the exit point of the subroutine to avoid state explosion（强制引擎在达到子程序出口点时停止，避免状态爆炸）. We then parse the memory of the resulting symbolic state, extracting all the indexes allocated for the binary’s subroutine offsets, effectively reconstructing the desired call table（然后解析产生的符号状态的内存，提取子例程偏移量分配的所有索引，有效的重构调用表）. Following these techniques we can resolve all branch targets in any arbitrary PRG binary, effectively reconstructing a 100% sound and complete CFG.

I/O operations and known functions matching（I/O操作和已知函数匹配）: We employ the I/O operation database to identify the instructions that result to reads/writes from/to physical I/Os（识别从物理I/O读写的指令）. For this task, we again perform symbolic execution with angr. In particular, we execute each subroutine individually, detecting read/write operations within the memory mapped I/Os ranges.（用angr单独执行每个子例程，检测内存映射I/O范围内的读写操作） We annotate the reverse engineering results to reﬂect these operations, assisting reverse engineers in identifying locations where the binary interacts with the physical environment（注释逆向工程结果来反映这些操作，协助逆向工程师识别二进制与物理环境交互的位置）. For matching in-binary subroutines to known library F/FBs, we calculate the signature of each subroutine, and query the known functions database for matches（匹配二进制文件内子例程到已知库F/FBs，计算hash进行匹配）. Whenever there is a match, the name of the subroutine is modiﬁed in ICSREF results to reﬂect its functionality per its library deﬁnition（修改子例程的名称）.

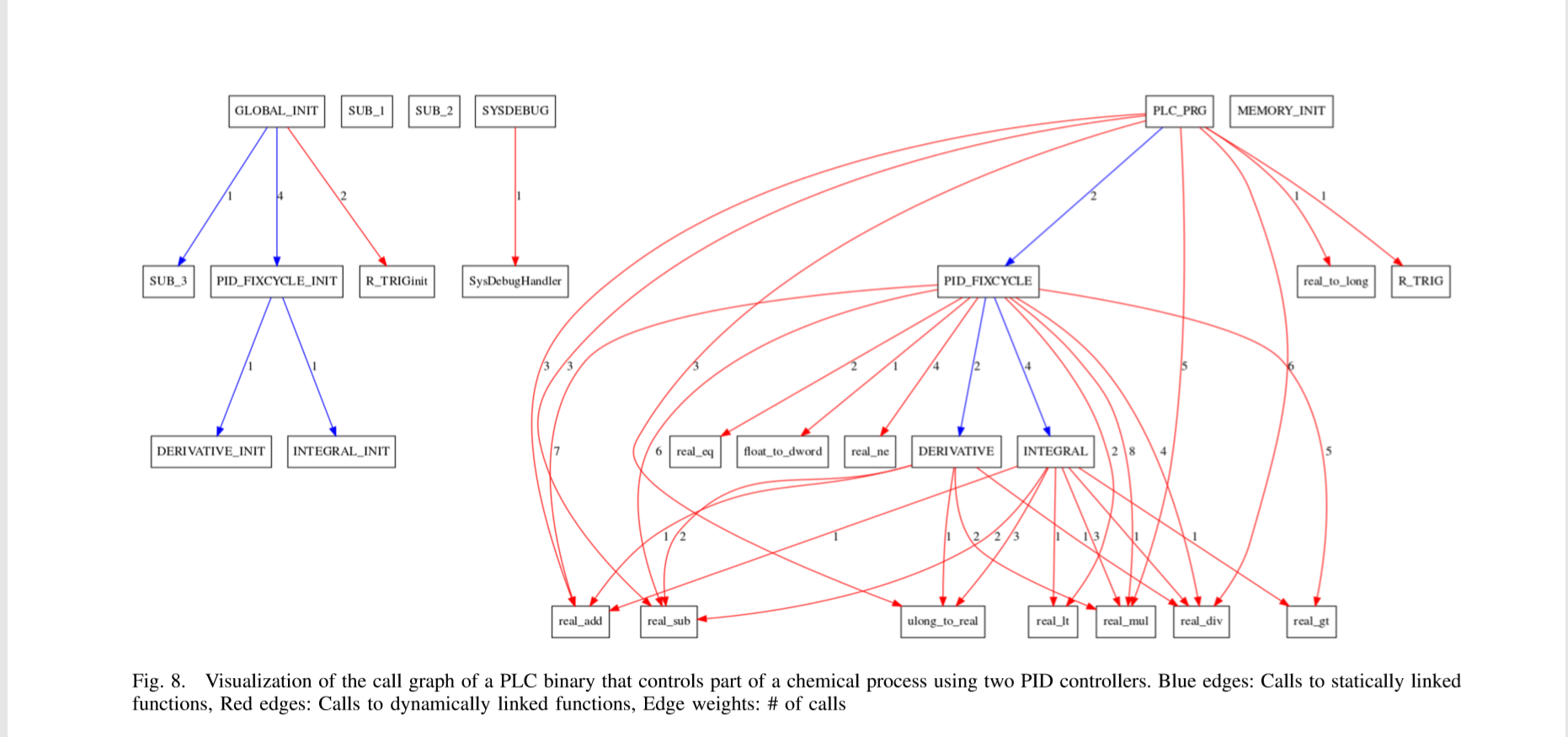
2) Function call arguments module: A typical reverse engineering step in dynamic analyses concerns extracting arguments passed to subroutines at call locations. This can aid in recovering lost source code, extract semantic-based information regarding the physical environment of the PLC, and allow process-aware dynamic payload generation. We implement function call argument extraction in ICSREF by developing a module that can extract the arguments passed to Proportional-Integral-Derivative (PID) function blocks as a proof-of-concept（该模块模块能提取传递到PID（比例-积分-导数）函数块的参数作为概念证明）. The PID block as deﬁned in the CODESYS Utility library is depicted in Fig. 7. We identify that arguments are passed to a subroutine on its local stack（栈传参）, and reconstruct this stack using angr. Taking into account that the arguments could be global variables/constants, we initially symbolically execute the Global INIT subroutine, where these values are deﬁned and set（参数可能是全局变量，需要执行Global INIT子例程）. Subsequently, we transfer execution to the caller subroutine and execute it until we reach a call to a function of interest (in this case PID). Finally, we query the resulting symbolic memory values on the local stack, effectively extracting the arguments that get passed to the PID function during its call. Although we develop an argument extraction module for PID functions, this module can be easily extended to extract the arguments of calls to any function for which the list of parameters and their data type are known.（我们将执行转移到调用者子例程并一直执行直到调用到感兴趣的函数，最终查询本地堆栈上得到的符号内存值）（上述步骤是获取参数的过程）



1. Binary modiﬁcation module: Introducing modiﬁcations to the binary under analysis enables a plethora of studies. To that end, we develop a binary modiﬁcation module for ICSREF. This module enables overwriting the contents of any location in a PLC binary (including both locations with code and data contents), with new user-speciﬁed values（用新的用户指定的值覆盖二进制文件中任意位置的内容）. Example uses include patching machine code instructions, changing branch targets, injecting executable code by overwriting “dead code”, and modifying the call arguments of critical functions, as they are extracted with the respective module. This module can also be used to introduce malicious modiﬁcations, enabling less sophisticated actors to craft malicious PLC binaries. In the interest of protecting critical infrastructure, although we will release all ICSREF analysis modules, the binary modiﬁcation module will not be disclosed publicly.（作者不会开源二进制修改模块）

To ensure that our modiﬁed PLC binaries are considered valid by the hardware PLC, we also investigate any employed ﬁle integrity mechanisms. In particular, CODESYS uses a checksum ﬁle (CHK) to ensure the integrity of binaries（CODESYS使用CHK文件确保完整性）. Thus, to generate valid modiﬁed binaries we need to reverse engineer this checksum algorithm. We achieve this by identifying the checksum algorithm to be an addition of all the bytes in a PRG binary, and incorporate CHK ﬁle generation in our binary modiﬁcation module.（校验过程：CHK文件存放PRG文件的所有字节累加）

1. Visualization module（可视化模块）: We develop a visualization module that provides intuitive representation of ICSREF results, addressing our usability design goal. The resulting call graph visualization after applying this module to a PLC binary is shown in Fig. 8. The graph is created using the DOT graph description language, where each subroutine and dynamically linked function is a node, and calls between them are edges（每个子例程和动态链接函数是一个顶点，调用是他们的便）. We add weights to the edges, representing the number of calls from a caller subroutine to its callee（权值为调用次数）. To differentiate between code contained in the binary and dynamically linked code, we use different colors for edges; for the former we use blue edges and for the latter red edges（蓝色线表示子例程，红色线表示动态链接函数）. Naming of unknown nodes follows the typical convention of sub\_<OFFSET>（未知节点的命名为sub\_<OFFSET>）, whereas nodes identiﬁed by the known functions matching technique use their library deﬁnition names to assist the reverse engineer（已知函数用它们在库中的定义）. For each node that is contained in the binary (i.e., not a dynamically linked function), we also create a hyperlink in the ﬁnal SVG image that links from a node to its corresponding disassembly listing for convenient traversal.（对于子例程，创建了超链接）



Considering our usability design goal, we develop all ICSREF modules in Python 2.7 allowing them to execute on cross-platform systems. The reverse engineer can also directly interact with the analysis and its results from a terminal interface created using the python cmd2 library.

## V. EXPERIMENTAL EVALUATION

To evaluate ICSREF both in terms of correctness as well as performance we carry out extensive experiments. For our experiments we utilize in-house PLC binaries and also collect real PLC programs and binaries from public code repositories. In this section, we present the experimental results of our analyses and the details of the database of sample PLC programs we create.（展示分析实验结果，和创建的样本PLC程序的数据库的细节）

1. Correctness evaluation （正确评估）

We verify the correctness of the platform-speciﬁc phase for CODESYS and the automated binary analysis phase by crosschecking and validating the results of analyzing a given PLC binary with ICSREF against its corresponding source code. This comparison allows a complete test since the source code that generated a binary under study is essentially a “golden copy” that contains all the information needed to evaluate the correctness of ICSREF results.（用源代码对比逆向结果进行测试）

1. In-house binaries: We perform a preliminary evaluation of the correctness of ICSREF using the 1704 PRG binaries, which we generated for the known functions database (see Section IV-A2). These binaries are excellent samples for preliminary tests, as we have the source code for all 1704 binaries, and in addition, the binaries have a very rigid structure, considering that they only include a call to one library F/FB by construction（只包含对一个库F/FB的调用？？？）. With this a priori knowledge we analyze all binaries with ICSREF, inserting automatic checks that compare the resulting CFGs with the expected CFGs stemming from the known source code structure.（比较CFG：生成的和源代码预期的比较） In all cases, ICSREF results matched the expected CFG, reconstructing a 100% complete and sound CFG. This provides a ﬁrst validation of the correctness of ICSREF results for rudimentary PRG binaries.

Subsequently, we utilize in-house developed binaries to evaluate ICSREF results when performing more complex analyses on real programs. We again compare and verify the correctness of ICSREF results against the expected functionality and CFG of the programs as it stems from their source code.

An example binary we use in this step is an IEC 611313 ST program targeting a WAGO 750-881 PLC. The program controls a small part of a chemical process, further described in Section VI. It contains calls to two PID control functions that execute in a ﬁxed time interval (using PID\_FIXCYCLE). The program also employs a digital input that acts as a trigger using rising edge detection (using R\_TRIG) that orchestrates the PLC program execution using an external signal. The resulting graph of the ICSREF analysis as generated by the visualization module is presented in Fig. 8.

To verify that ICSREF results are correct, we perform a line-by-line comparison of the results against the program’s source code, verifying that they match. Here, in the interest of space, we only focus on high level subroutines and their calls. As we observe from Fig. 8, the nodes and their edges as detected by ICSREF completely agree to the above description of the program operations. The main PLC\_PRG subroutine calls the PID\_FIXCYCLE function block twice, which in turn calls the DERIVATIVE and INTEGRAL functions; note that each call to PID\_FIXCYCLE calls DERIVATIVE once and INTEGRAL twice, something we derive by examining the source code of PID\_FIXCYCLE from its corresponding library ﬁle. PLC\_PRG also calls R\_TRIG for detecting the external triggering signal. The Global INIT subroutine initializes all the required subroutines. Several other dynamically linked function calls are made from the main function and its callees; these are either made by the imported library FBs for performing their calculations, or carry out scaling/formatting of environment values, something we veriﬁed by analyzing in depth both the developed source code, as well as all imported library F/FBs source code.

2) Binaries from public code repositories: In addition to in-house developed binaries, we collect real programs and binaries developd using CODESYS from online, public repositories on GitHub. These programs are developed by different authors, target various PLC devices, and control different, disparate physical environments. This makes them particularly appropriate for testing our framework, as their diversity can rigorously test the correctness and generality of ICSREF.

Beyond ICSREF correctness evaluation, by collecting and consolidating PLC programs and binaries we create a samples database. Our database contains both the source code as well as the corresponding binaries for industrial applications of various complexities, enabling various studies by providing a repository of diverse PLC programs. An immediate beneﬁt of our database is that it provides benchmark programs that enable a thorough evaluation of methods and techniques that operate at the source code level, the binary level, or in between (as ICSREF does). We leverage this database for exactly this purpose in this paper. Furthermore, our database enables comparative studies with a common set of benchmark programs, and provides useful and realistic PLC program source code for educational purposes, lowering the entry bar to PLC programming and process engineering.

We construct speciﬁc search queries using the GitHub advanced search options to discover public code repositories containing CODESYS source code project ﬁles. This is not straightforward; while these ﬁles contain the source code of PLC applications, they are non human-readable ﬁles. As such, the GitHub search engine does not index them, and it cannot perform searches within their contents. We overcome this limitation by leveraging the auxiliary ﬁles that support CODESYS project ﬁles (\*.pro) in order to pinpoint and collect our database samples. When developing a PLC program with CODESYS, the IDE creates several auxiliary ﬁles, such as compile information ﬁles, download/reference ﬁles,initializationandlogﬁles,andsymbolﬁles.Alltheseﬁles are non-text ﬁles, except the \*.SYM symbol ﬁles. The symbol ﬁles comprise of ASCII characters and contain information regarding the version, project Id, checksum, and size of a project. Since they are indexed by GitHub, we use keywords present in these ﬁles (e.g., “ProjectId”), in combination with a search for ﬁles with a SYM extension, and ﬁnd the public code repositories that contain CODESYS projects.（用文件的后缀名搜索项目）

In total, we identify and download 471 CODESYS project ﬁles from 127 repositories of 55 users across GitHub using the search terms described above. For 69 out of the 471 project ﬁles, the repositories also contained the corresponding PRG binaries. By examining the strings contained in the project ﬁles we extract the target PLC details, identifying the vendor and target PLC model. Table III shows the target PLC device vendors for the downloaded project ﬁles. From the table we observe that the majority of programs (68%) are developed for WAGO PLCs. This can be attributed to two factors. First, WAGO PLCs are popular for small-to-medium installations which are also more likely to share their process engineering programs publicly. In contrast, large corporations employing devices from other vendors are arguably much more protective of their source code. As their code likely controls sensitive industrial deployments, making the code publicly available could divulge corporate secrets and intellectual property. Second,235 out of the 471 projects we downloaded belong to a single organization, which provides software and control solutions for the energy sector based on WAGO PLCs.