ICSQuartz: Scan Cycle-Aware and Vendor-Agnostic Fuzzing for Industrial Control Systems

Corban Villa*, Constantine Doumanidis*, Hithem Lamri*,
Prashant Hari Narayan Rajput† and Michail Maniatakos*
*Center for Cyber Security, New York University Abu Dhabi, Abu Dhabi, UAE
†InterSystems, Cambridge, MA, USA

***Abstract*—Industrial Control Systems (ICS) ensure the automation and safe operation of critical industry, energy, and commerce processes. Despite its importance, ICS code often cannot be evaluated as rigorously as software on traditional computing platforms, as existing code evaluation tools cannot readily interface with the closed ICS ecosystem. Moreover, the use of domain-specific languages, the lack of open and extensible compilers, and the deficiency of techniques developed for ICS-specific nuances, among other challenges, hinder the creation of specialized tools. This paper addresses these challenges by introducing ICSQuartz, the first native fuzzer for IEC 61131-3 Structured Text (ST), a standardized Programmable Logic Controller (PLC) programming language. Native support eliminates the necessity of any vendor or architecture-specific requirements.**

ICSQuartz outperforms the fastest state-of-the-art fuzzers in the ICS space by *more than an order of magnitude in executions per second*. In addition to natively fuzzing ST code, we introduce novel mutation strategies to ICSQuartz that uncover vulnerabilities due to the scan cycle architecture of ST programs–a nuance that traditional fuzzers do not consider. Using ICSQuartz, we perform the first large-scale fuzzing campaign of real-world ICS libraries, resulting in multiple vulnerability disclosures and bug fixes. In addition to vulnerabilities, ICSQuartz discovered a bug in an open-source ST compiler. These findings underscore the imperative impact of ICSQuartz in the ICS domain.

I. Introduction

Industrial Control Systems (ICS) represent a collection of systems, networks, and associated connected equipment that operate in tandem to control, monitor, and automate physical processes. ICS control processes in a wide array of fields, including manufacturing (e.g., operating assembly lines), energy (e.g., geographically distributed operation of electric smart grid), mining (e.g., material handling), and critical infrastructure (e.g., automating water treatment facilities).

Programmable Logic Controllers (PLCs), an integral part of ICS, are industrial computation platforms equipped with specialized hardware and peripherals and built for robust and reliable operation in harsh industrial conditions. In these environments, it is common for manufacturers to equip PLCs with firmware that utilizes standard operating systems such as RTLinux and VxWorks. These devices follow a periodic mode of operation known as the scan cycle: The PLC captures inputs from the monitored physical process (e.g., temperature readings, control signals, machinery status, etc.), processes these inputs based on its programming, and utilizes the outputs to control actuators that influence the underlying process. Process engineers program PLCs by writing control logic code using domain-specific programming languages, such as Structured Text (ST), that are defined under the IEC 61131-3 standard [1]. This code is then compiled using vendor-supplied proprietary compilers to non-standard formats. Compiled control programs are then executed by purpose-built software stacks, known as runtimes, which run on the PLC.

The transformation of the industrial landscape by the movement dubbed the 4th Industrial Revolution [2], saw the operational technology (OT) space merge with solutions and practices from the Information Technology (IT) space. This development has seen ICS devices gain increased flexibility, support greater programmability, receive augmented with advanced real-time monitoring features, and numerous other capabilities that extensively benefited adopting industries. Furthermore, ICS vendors have embraced open source software (OSS) solutions to support their ecosystems, reducing resources spent reimplementing and securing software, such as FTP servers, SSH clients, or even complete firmware stacks [3]. These developments have led to a paradigm shift in the security landscape of the ICS code, requiring that components previously thought of as isolated, such as the IEC 61131-3 programs [1], be rigorously evaluated and secured.

Given their critical positioning and role, ICS are a prominent target for malicious parties. Historically, weaknesses in ICS software have led to events such as Stuxnet [4], allowing adversaries to compromise Iran's nuclear infrastructure by injecting malicious code into PLCs, thus impairing close to 20% of their centrifuges. ICS-specific malware was also discovered in 2016 by the FLARE team, which identified IRONGATE [5], a malware family built to manipulate industrial processes operating on Siemens' control system environment. Another incident in 2019 saw threat actors successfully enter the US Coast Guard network, forcing the facility to shut down operations for more than 30 hours [6]. In 2022, the European Union Agency for Cybersecurity (ENISA) predicted an increased risk to OT systems, due to a rise in sophisticated state-level threats and malware targeting ICS [7]. The following year,

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240795
www.ndss-symposium.org

ENISA remarked this coming to fruition with the discovery of Industroyer2, COSMICENERGY, and CRASHOVERRIDE, and called for proactive efforts to secure OT and ICS [8].

Therefore, identifying faulty programming and vulnerabilities in PLC code is a topic of great interest to the research community. The work carried out in this domain has considered various directions, such as model checking [9], automated tests and generation of invariants for PLC code [10, 11], formal verification of control logic [12, 13, 14], and reverse engineering of control binaries [15]. These methods provide important insights and safety guarantees; however, they focus on PLC code that strictly follows the IEC 61131-3 standard and do not consider the effects of the underlying runtime, its libraries, and its dependencies.

PLC vendors are expanding their software stacks to support an increasing amount of features [3], ranging from simple language tweaks, libraries, and visualization systems, to fully fledged machine learning solutions [16]. These increased capabilities of PLCs necessitate that PLC code security evaluation takes into consideration the a refined threat model. An obvious remedy would be to resort to mature tools from the IT space: Dynamic and static analysis, fuzzers, and symbolic execution-based tools. However, application of these tools to the ICS domain is not straightforward due to challenges stemming from the domain-specific languages, closed-source compilers, non-standard binaries, and proprietary protocols. Efforts to bridge IT with OT space tools need to overcome significant obstacles [17, 18, 19], while the lack of vulnerable ICS code corpora deters the creation and evaluation of such tools.

In this work, we introduce ICSQuartz, a novel framework developed for fuzzing ST code. Our approach utilizes an open-source compiler to enable LLVM code instrumentation during the compilation of ST programs, which is notably vendor-agnostic and platform-independent. This enables us to conduct native fuzzing for ST programs, encompassing scan cycle fuzzing, coverage-guided fuzzing, and vulnerability localization. These improvements to state-of-the-art solutions manifest through multiple vulnerability disclosures. In summary, our contributions are the following:

- We introduce ICSQuartz, the *first native fuzzer* for the IEC 61131-3 Structured Text language. ICSQuartz is vendor-agnostic and platform-independent, surpassing existing solutions in performance and capability, incorporating advances from non-OT state-of-the-art fuzzers.
- We develop new ICS-specific scan cycle fuzzing components, which discover vulnerabilities unique to ICS environments, and extend ICSQuartz to support it.
- We demonstrate the impact of ICSQuartz by producing multiple vulnerability disclosures and bug fixes in widely distributed ST libraries. We discovered, to the best of our knowledge, *the first ST-related CVE*. At the same time, ICSQuartz also discovered a bug in an ST compiler which was leading to generation of vulnerable ST programs.
- ICSQuartz and developed benchmarks have been artifact-evaluated as reproducible (Appendix A) and are available: https://github.com/momalab/ICSQuartz.

II. Preliminaries

A. IEC languages

Code written for industrial control systems has been systematized by the International Electrotechnical Commission through the formation of the IEC 61131 international standard [1]. IEC 61131-3, the third component of the standard, defines an ensemble of five interoperable, domain-specific programming languages to be used with PLCs. The IEC 61131-3 family defines graphical languages, including Ladder Diagram, Function Block Diagram, and textual languages, such as Structured Text. For this work, we elect to use Structured Text (ST), which closely resembles traditional imperative programming languages and is the most flexible language in the standard for writing complex control functionality code.

B. The PLC Software Stack

The typical PLC software stack is quite complex, involving many components, including an Integrated Development Environment (IDE) for code development, compilers for IEC 61131-3 languages, a runtime that executes on the PLC, and special communication protocols. Due to the increasing complexity of the PLC software stack, and the costs associated with its maintenance and development, many ICS hardware vendors have adopted and customized readily available software stacks, such as Codesys [20].

Projects are written using a combination of IEC 61131-3 languages and are compiled using specialized proprietary compilers provided by ICS vendors. These compilers differ from commodity solutions like the GNU C Compiler (`gcc`) and the Clang compiler (`clang`), as they prioritize predictability, robustness, and stability, which are of paramount importance in OT, over performance and optimization, which are often preferred in IT. The compilation output is typically a binary in a proprietary format defined by the vendor. ICS control logic binaries can either be bytecode that is then interpreted during execution or assembly instructions.

The IEC 61131-3 execution model diverges from traditional execution models, relying on scan cycles. In the traditional IT domain, typical programs have a standardized lifecycle: 1) An initial setup sequence by the operating system to allocate memory and resources. 2) Subsequent execution of instructions as a process. 3) System resources are released as the process terminates. In the case of PLCs, programs run as proprietary executables on top of a vendor runtime process. The runtime is responsible for executing the IEC 61131-3 program as part of the scan cycle at a predefined frequency configured by the operator [1]. In IEC 61131-3 programs, variables are initialized once by the runtime and are then mutated by program logic until the PLC is shut off.

C. Vulnerabilities in PLC Software

Previous work has explored how ST, similar to C and C++, is a memory-unsafe language and is inherently vulnerable to various memory-exploitation bugs [21]. A common example of this is arrays with an absence of bound checks prior to reading from memory, even when indexes may be greater than the size

of the array or are negative values. Other problems stem from the language's permissive rules regarding the use of pointers, which can be easily manipulated and reassigned through code.

Libraries provided by vendors also introduce access to memory-unsafe functions, such as `SysMemCpy`, `SysStrCpy`, and `SysMemSet`, which have been determined to not perform bound checks before modifying memory [17]. Unlike C or C++, the ST language specification does not include a system for dynamic memory allocation, such as with `malloc` or `new`. Instead, as PLC programs are intended for repeatable and consistent execution, the programs must rely only on stack memory. Some vendors have introduced an extension to the ST language, which introduces support for dynamic memory allocation [22]. These non-standard language extensions further increase the possibility of memory-related vulnerabilities and reinforce the importance of fuzzing to identify security vulnerabilities proactively.

D. RuSTy Compiler

Although proprietary implementations remain largely prevalent within the IEC 61131-3 development space, RuSTy is a recently introduced open-source ST compiler, written in the Rust programming language [23]. RuSTy implements the IEC 61131-3 Structured Text language specification, allowing programs written to run on PLCs to be compiled for other platforms and architectures. It manages this by leveraging the Low-Level Virtual Machine (LLVM) compiler framework [24], which provides a robust compiler backend throughout the low-level stages of program compilation.

III. Problem Formulation

A. Limitations of ST Evaluation

Proprietary Compilers: A core inhibitor to the application of modern fuzzing tools on ST is the prevalence of proprietary compilers. Proprietary PLC compilers are often built into the vendor IDEs and offer minimal customization. Available features are limited, and none of these compilers, to our knowledge, offer any code instrumentation features that are compatible with modern security fuzzing tools.

Proprietary Executable Formats: The binaries produced by said proprietary compilers are also, in most cases, in proprietary formats. Unlike compilers used by most UNIX operating systems, such as `gcc` and `clang`, which produce standard Executable and Linkable Format (ELF) binaries, PLC binaries are most often black-box binaries. Though some previous works have managed to reverse-engineer some of these binary layouts after significant effort [15, 21], the non-compliance with the ELF standard makes running these applications on any system, without the assistance of the vendor runtime, non-trivial. Additionally, beyond reverse-engineering the binary layouts, many of these programs depend on components of the proprietary runtime for core components to function, such as the delivery of sensor inputs to the program.

Proprietary Libraries: Vendors often provide access to extensive collections of first-party and third-party libraries that fulfill a variety of purposes, from networking operations to reading and writing to the filesystem to performing basic string operations. In most cases, these libraries are pre-compiled and only include enough code to define function interfaces. These proprietary libraries present another challenge when attempting to fuzz ST for security vulnerabilities: 1) White-box fuzzing techniques, which typically rely on compile-time instrumentation, are not possible. 2) Black-box fuzzing techniques are significantly hindered by proprietary executables and non-standard memory layouts.

B. Limitations of Related Work

Non-Standard Fuzzing Interface: A fundamental limitation of previous work is the non-standard fuzzing interfaces used to deliver inputs. Creative solutions have been developed by researchers, including hijacking the `KBUS_CYCLE` thread of the Codesys Runtime, which is responsible for delivering program inputs at every execution. While this technique enables fuzzing ST programs, it often experiences unreliable input delivery due to scan cycle synchronization issues [17].

Another solution, introduced in FieldFuzz [18], delivers program inputs over the network by leveraging the proprietary network protocol used to communicate with the Codesys Runtime. Two unfortunate side-effects of this delivery mechanism are: 1) Hard-coded memory offsets for each program input, which must be manually located by the operator, and requires that address space layout randomization (ASLR) be disabled. 2) Requiring unauthenticated access to the runtime.

Although the network-based input delivery provides faster and more reliable inputs than the `KBUS_CYCLE` technique, both techniques are non-standard and highly dependent on vendor runtime implementation details. Crucially, both techniques present a much higher barrier to entry than standard methods and suffer from additional overhead compared to traditional IT fuzzers, which typically pass inputs directly through system memory. In addition, adapting state-of-the-art security fuzzers to these input delivery channels presents significant technical challenges.

Non-Standard Coverage Mechanisms: Another fundamental limitation of related work is non-standard mechanisms to track code coverage in executing programs. One method explored previously by researchers replaces `NOP` (no-operation) assembly instructions in program binaries with logging instructions to track execution paths [17]. Another method uses a dynamic instrumentation toolkit to monitor executing functions and relevant segments in memory and extract coverage feedback for the fuzzer. While effective, the authors noted a significant resource consumption as a result of this dynamic instrumentation [18]. Moreover, both mechanisms require fuzzers to be specifically adapted to receive coverage data through these unique channels.

Vendor Lock-In: State-of-the-art fuzzing tools in the ICS domain are often vendor-dependent, relying on leveraging proprietary features of the Codesys Runtime, which are unfortunately subject to change at the discretion of the vendor. Case in point, recent changes in the Linux kernel have broken the input delivery mechanism leveraged by ICSFuzz [17]. Further

Work	Control Application										Misc.			
	Support	Vuln. localization	Input delivery	Vendor-agnostic	Scan cycle control	Coverage feedback	Crash monitoring	Harness generation	Address sanitizer	Scalable	Unique crashes	PLC independent	Arch. independent	Runtime fuzzing
ICSFuzz [17]	●	○	◐	○	○	◐	◐	○	○	○	○	○	○	◐
FieldFuzz [18]	●	○	◐	○	◐	◐	◐	○	○	◐	○	○	◐	◐
ICSQuartz	●	●	●	●	●	●	●	●	●	●	●	●	●	○

TABLE I: **ICSQuartz vs. State-of-the-Art ICS Fuzzers.** Full support indicated by ●. Partial or suboptimal support indicated by ◐. No support indicated by ○.

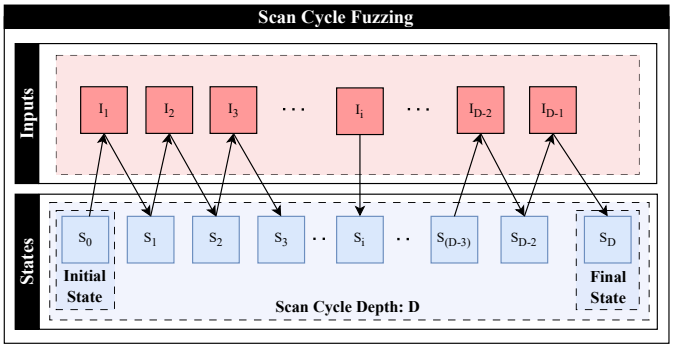

Fig. 1: **Scan Cycle Execution Paradigm.** New inputs consecutively execute on the prior state, S_{i-1}, producing S_i.

complicating, the firmware binary, which includes this older Linux kernel, is no longer available in the vendor repositories, rendering the extensive work currently non-functional.

Insufficient Memory Bug Detection: Fine-grain detection of potential memory vulnerabilities is crucial for effective fuzzing. Current ICS fuzzing tools depend on the operating system to detect out-of-bounds reads or writes through segmentation fault (`SIGSEGV`) exceptions [17], or through rule-based dynamic analysis which enforces separation of program and runtime memory-isolation [21]. Memory bugs that attempt to read or write within valid program memory, which can be more dangerous as those that do reach beyond valid memory, will not be detected by current ICS fuzzing tools.

Missing Scan Cycle Evaluation: The scan cycle execution model is unique to IEC 61131-3 programs. Prior fuzzing tools do not take this ICS-specific novelty into account in their designs. Although ICSFuzz [17] inherits stateful execution as it operates on a physical PLC, the fuzzer does not have control or knowledge of the program state while executing. Similarly, while FieldFuzz [18] does provide limited program-state control capability (start/stop execution), it also considers each execution absolute. Crashes in both of these solutions lack sufficient information to record inputs leading to a crash, rendering reports non-reproducible.

Manual Effort Required: The manual effort required to setup and configure current tools presents another barrier to entry for robust evaluations. ICSFuzz[17] requires a physical PLC running the appropriate firmware and Codesys Runtime version. It additionally requires a compatible Windows development machine to run the Codesys IDE and monitor the fuzzing process. Lastly, it requires manual crash monitoring, as the PLC must be manually restarted when it encounters bad input. FieldFuzz [18], on the other hand, requires an operator to capture and dissect PLC traffic using Wireshark to generate an initial input corpus for a sample program [18].

Both state-of-the-art solutions require operators to have significant familiarity with the tools themselves and to invest a considerable amount of time to configure and monitor the fuzzing process. Lastly, both projects lack a mechanism for identifying unique crashes and instead record all crashes encountered, making crash investigation a laborious undertaking.

Lastly, once one or more crashes have been produced by ICSFuzz [17] or FieldFuzz [18], the operator must manually triage the proprietary ST program in the vendor IDE to localize the vulnerability to a specific portion of problematic code.

Summary of Comparison: Table I synthesizes the comparison of our work, ICSQuartz, with prior ICS fuzzers. Although some fuzzers are capable of fuzzing portions of the proprietary runtime [17, 18], we consider this out of the scope of ICSQuartz. Specifically, this is because vendor runtimes are implemented in traditional programming languages (e.g. C/C++), and can be most effectively evaluated by vendors with white-box access to the source. Instead, ICSQuartz focuses exclusively on providing a robust evaluation of ST programs.

IV. ICSQuartz

To resolve the outstanding technical challenges related to the robust security evaluation of ICS, we introduce ICSQuartz, a vendor-agnostic and platform-independent ICS-specific fuzzer, equipped with both scan cycle aware and coverage-guided fuzzing techniques.

A. Scan Cycle-Aware Fuzzing

Figure 1 depicts the scan cycle execution model, a core component of IEC 61131-3 program execution with the potential to introduce new and unexpected vulnerabilities, specific to ICS programs.

Scan cycles are defined to execute at regular intervals: Reading inputs from sensors, executing control logic, and writing outputs. In this pattern, the IEC 61131-3 program is executed at cycle time frequency [1], often measured in milliseconds, which we denote as τ. In the first execution, the program will inherit the initial variable state, defined by the programmer, which we denote as state S_0. After τ ms, the program will execute once again, operating on values from state S_0 and producing state S_1, which will then be inherited by the next execution. To calculate any program state S_i, which is defined as the program state after $i \times \tau$ ms, the previous state S_{i-1} must first be computed.

Suppose that the execution state at S_n is responsible for a memory vulnerability. Reconstructing the crash requires the set of all states in S:

$$S = \{S_0, S_1, S_2, \ldots, S_n\}$$

To extend ICSQuartz to support scan cycle-execution fuzzing, we isolate program inputs from program state, enabling us to cycle through many program inputs while preserving integrity of the IEC 61131-3 program state.

The number of states simulated and explored by ICSQuartz, known as scan cycle depth, is denoted as D, and is equal to the cardinality of the set of states observed, S.

A novel component of ST programs is a unique execution pattern, often featuring cyclic control flow through initialization, incremental computation, completion and reset phases [25, 26, 21]. The number of scan cycles required for an exhaustive execution is not necessarily deterministic, and rather varies as incoming sensor inputs are continually received and processed.

A naive fuzzing strategy to explore these types of programs may invoke a constant scan cycle depth: Executing the program across D inputs, after which the state will be cleared to default values and a new simulation can begin. This mechanism, while simple to implement, introduces two key drawbacks. **1)** A insufficiently high scan cycle depth D may mask the presence of vulnerabilities which manifest beyond D. **2)** A scan cycle depth D which is configured as too high may spend a significant proportion of executions on programs which are longer processing new inputs. This naive strategy could potentially be introduced to FieldFuzz [18], yet cannot be developed further due to the black-box architecture. As ICSFuzz [17] does not feature state control, this would not be possible to implement.

Adaptive Scan Cycle Execution: ICSQuartz, rather than relying on a constant scan cycle depth D, addresses this challenge by actively monitoring for programs which are no longer processing inputs and require a state reset immediately. We achieve this functionality by introducing a fuzzer observer [27] which copies the program state before and after execution at the byte-level, can be dynamically configured by an operator in the fuzzer harness. It subsequently compares the two states to determine whether the inputs had any affect on the program. We denote the cases where the program state does not change as stale scan cycles, which ICSQuartz reacts by invoking a state reset. The number of stale states required to invoke a state reset defaults to 1, but is configurable by the operator for more sophisticated contexts.

As it is possible that a ST program may include an execution branch which never terminates, we include a maximum scan cycle depth parameter for these edge-cases. This mechanism forces a programs to reset regardless of changes to their state, preventing executions which would otherwise never terminate.

In order to provide introspection into the scan cycle fuzzing process, we instrument ICSQuartz to report several metrics in the fuzzer output corpus, including: **1)** The number of stale scan cycles encountered. **2)** The respective scan cycle for coverage and crash reports. **3)** All inputs delivered in the given scan cycle context, allowing a comprehensive reconstruction of the vulnerable state S_n and any previous states, S_i, which led to the vulnerable state.

Library	Vendor Function	Target Function	Source Lib.
SysMem	`SysMemSet`	`memset`	`glibc`
	`SysMemCpy`	`memcpy`	`glibc`
	`SysMemMove`	`memmove`	`glibc`
MemUtils	`MemSet`	`memset`	`glibc`
	`BitCpy`	`bitcpy`	(custom)
SysFile	`SysFileOpen`	`fopen`	`glibc`
	`SysFileRead`	`fread`	`glibc`
	`SysFileWrite`	`fwrite`	`glibc`

TABLE II: **Vendor-Specific Compatibility Layers.** Proprietary vendor functions are mapped to the target functions, displayed as `glibc` functions or custom implementations.

B. ICSQuartz New Mutation Strategies

In order to tailor ICSQuartz for ST program nuances, we introduce a number of specialized mutation strategies, which are then combined with existing state-of-the-art mutations from AFL++ [28].

Scan Cycle Unblocking: As discussed previously, many ST programs feature cyclic control flow patterns, which must be reset by providing specific inputs [25, 26, 21]. To that end, our scan cycle unblock algorithm attempts to simulate this type of sensor input by selecting a random byte from the input and replacing it with its respective initial value. These initial values do not need to be configured manually by the operator, and are instead retrieved from the RuSTy [23] compiler.

ST Input Shaping: Contrasting to traditional IT programs which feature inputs of variable lengths, ST programs most often receive inputs from a static configuration of sensors. Each input has a hard-coded length, and no dynamic memory allocation is allowed [1]. The input shaping mutation strategy retrieves the program input size from the RuSTy [23] compiler, ensuring that inputs are the appropriate size for the program. While input size constraints could alternatively be enforced in the fuzzing harness, implementing it as a mutation strategy ensures that all inputs are executable by the program.

These ST specific mutation strategies and scan cycle adaptation complement the existing state-of-the-art mutation algorithms provided by AFL++ [28].

C. Vendor-Agnostic and Platform-Independent Fuzzing

Compatibility Layers: Proprietary vendor libraries still posed a challenge to a complete evaluation of many ST programs, which are leveraged in real-world control programs and in state-of-the-art benchmarks [17, 21]. The complexity of these vendor libraries varies significantly by function. Fortunately, many of these functions are wrappers named after the underlying `libc` functions invoked (e.g. `SysMemCpy` calls `memcpy`). Higher complexity functions, such as those which abstract network functionality through a series of system calls

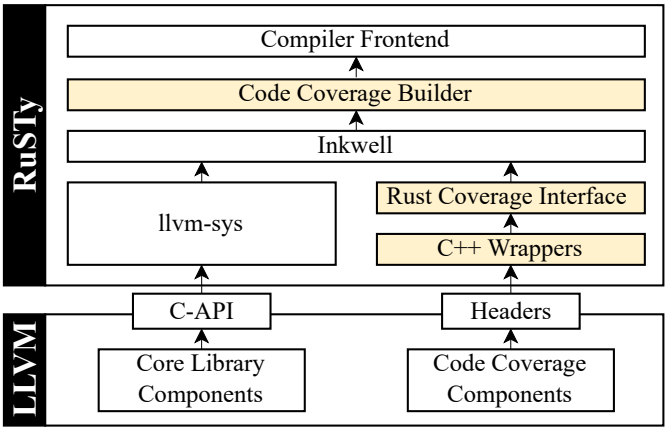

Fig. 2: **ICSQuartz Code Coverage Architecture.** Implementation of code coverage components in RuSTy for ICSQuartz. Highlighted components indicate our contributions.

(e.g `SysSockSelect`), may require more thorough reverse-engineering efforts leveraging other binary analysis tools to fully understand.

To enable compilation of benchmarks in related work, we create a compatibility layer comprised of the proprietary ST function interfaces, which are mapped to their underlying `libc` counterparts, or to our own C-based implementations, which are compiled and linked against. Table II includes samples of function mappings, provided through our compatibility layer. This compatibility layer introduces a certain margin of error for vulnerabilities to be either introduced or to be concealed, in cases where our implementation does not perfectly match that of the vendor library. This concern can be alleviated through the cross-validation of potential security vulnerabilities on a physical PLC testbed.

Structured Text Code Coverage: As discussed, code coverage instrumentation is crucial in enabling coverage-guided fuzzing. To perform a comprehensive evaluation of ST targets, we introduce LLVM code coverage support to the RuSTy compiler, enabling compatibility with state-of-the-art fuzzing techniques. This work further narrows the gap between traditional IT fuzzing and ICS fuzzing, addressing a fundamental obstacle that earlier hindered the use of traditional fuzzing tools on ST.

LLVM code coverage instrumentation introduces source-based metadata and control flow tracking, such that precise sections of code can be identified during execution. The coverage instrumentation includes injecting counters and increment calls (i.e. `llvm.instrprof.increment`), which are invoked every time functions and code branches are reached, allowing fuzzing algorithms to understand which inputs explore new branches and determine how much of the program paths have yet to be explored. Due to the tight integration of source code metadata and branch counting, the majority of this instrumentation must be performed in the RuSTy compiler itself, and cannot be implemented as an LLVM Intermediate Representation (IR) pass.

Figure 2 provides an architecture diagram that visualizes

Algorithm 1 ICSQuartz Code Coverage Algorithm.

```
CONTROL_FLOW ← {IF, CASE, FOR, WHILE, REPEAT}
procedure INSTRUMENT(functions)
    for all function ∈ functions do
        name ← function.name
        span ← function.span
        // Parent counter (C_P) tracks function executions.
        C_P ← CREATECOUNTER
        // Invoke the DFS algorithm.
        GENCOVERAGE(function.ast, C_P)
        STOREFUNCTIONRECORD(name, span, C_P)
    end for
end procedure
function GENCOVERAGE(ast, C_P)
    for all node ∈ ast.nodes do
        if node.type ∉ CONTROL_FLOW then
            continue
        end if
        // False counter (C_F) is initially set to C_P and cascades
        // and as new branches are recorded.
        C_F ← C_P
        if node.type ∈ {IF, CASE} then
            for all branch ∈ node.branches do
                C_T ← CREATECOUNTER
                // C_F is recursively calculated from parents.
                C_F ← CREATEEXPRESSION(C_F − C_T)
                STOREREGION(branch.span, C_T, C_F)
                GENCOVERAGE(branch.body, C_T)
            end for
        else if type ∈ {FOR, WHILE, REPEAT} then
            for all block ∈ node.blocks do
                C_T ← CREATECOUNTER
                C_F ← CREATECOUNTER
                STOREREGION(block.span, C_T, C_F)
                GENCOVERAGE(block.body, C_T)
            end for
        end if
    end for
end function
```

why introducing the LLVM code coverage to RuSTy is non-trivial. As the code coverage components of LLVM are not exposed through the official C++ API, it is instead required to interface with these functions through the inclusion of specific header files and subsequently linking against LLVM. The architecture depicted exposes the necessary interfaces to RuSTy, which leverages the Rust-based Inkwell library to provide a layer of type-safety from the Foreign Function Interfaces (FFIs) which facilitate direct Rust and C++ interoperability [29]. We also extend Inkwell to enable this integration.

Once the appropriate LLVM code coverage functions were made accessible to RuSTy, coverage instrumentation was implemented using a depth-first-search algorithm to traverse the ST program Abstract Syntax Tree (AST), which is shown in Algorithm 1. This algorithm records where `llvm.instrprof.increment` calls should be placed, and introduces both execution counters and counter expressions. While counters are simple incrementing data structures, counter expressions are calculated dynamically based on simple counters, optimizing performance and storage usage.

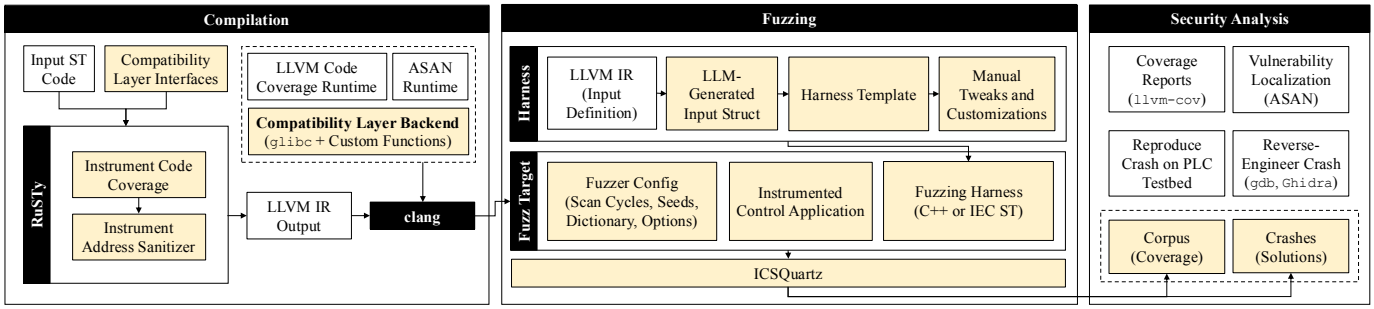

Fig. 3: **ICSQuartz Fuzzing Pipeline: 1)** Programs are compiled with fine-grain security instrumentation. **2)** Automatic harnesses are used by ICSQuartz to fuzz. **3)** Crashes are triaged for vulnerabilities. Highlighted components indicate our contributions.

```
+@__llvm_coverage_mapping = private constant {...
+@__profc_prg = private global [1 x i64] zeroi...
+@__profd_prg = private global { i64, i64, i64...
...
define void @prg(%prg* %0) #0 {
    entry:
        %a = getelementptr inbounds %prg, %prg...
        %b = getelementptr inbounds %prg, %prg...
+       %pgocount = load i64, i64* getelementp...
+       %1 = add i64 %pgocount, 1
+       store i64 %1, i64* getelementptr inbou...
        store float 1.500000e+00, float* %b, a...
        %load_b = load float, float* %b, align 4
        %2 = fptosi float %load_b to i16
        store i16 %2, i16* %a, align 2
        ret void
}
```

Listing 1: **ICSQuartz Code Coverage Instrumentation.** IR sample of injected coverage counters and data structures.

Listing 1 includes a simplified sample of the LLVM code coverage introduced to a small ST program. This coverage instrumentation feature will be merged into RuSTy for others to continue research in this area [23].

Address Sanitizer: A core feature of state-of-the-art fuzzing tools is the ability to detect memory bugs at a fine-grain level, beyond bugs which cause segmentation faults (`SIGSEGV`). The Address Sanitizer (ASAN) LLVM IR pass became the ideal candidate to introduce higher precision memory bug detection into RuSTy, as it boasts a comparatively low overhead to other solutions, with an approximate 73% decrease in execution performance [30]. ASAN is the primary address sanitizer integrated into multiple robust compilers, such as `gcc`, `clang`, and `rustc`.

To incorporate these advances from traditional IT fuzzing, we extend RuSTy to support the industry-standard ASAN instrumentation. This is accomplished through additional instrumentation across the RuSTy frontend and by the invocation of the the ASAN LLVM IR pass to complete the ASAN implementation. This provides ICSQuartz the ability to discover more sophisticated vulnerabilities, which are fundamentally not detectable by previous state-of-the-art ICS fuzzers [17, 21].

Input Delivery: Reliable and low-latency input delivery is a key factor that contributes to fuzzing performance. Program entrypoints in RuSTy, defined as functions, use inputs and outputs defined in continuous C-like structures, passed to the program as a memory address. The program reads inputs from this structure, executes the control logic, and writes outputs, which can be read after the execution concludes. This enables ICSQuartz to send program inputs directly as system memory to a function, rather than over the network or by taking over a runtime thread, as previous state-of-the-art fuzzers [17, 18].

LLM-based Harness Generation: The fuzzing harness can be written in C++ or ST. As previous works required significant manual effort before fuzzing could commence, we developed a tool to automatically generate a C++ fuzzing harness for any provided program. This harness generation tool leverages the program's IR as a ground truth for byte-level inputs and outputs, which are passed to a Large Language Model (LLM) to predict a C-struct, which models the layout of the original program. This is then inserted into a template fuzzing harness, to be customized further as needed. Sample prompts used for the LLM-generated harnesses can be found in the Appendix (Listing 3, 4).

Vulnerability Localization: The code coverage instrumentation and address sanitizing allow ICSQuartz to identify precisely which instruction executing in a program caused a crash without the need for analysis with an additional crash triage process as previous ICS-specific fuzzers required. In addition, ICSQuartz can identify whether a similar crash has been reported before and only report unique crashes. Further, once a crash is encountered, mutation strategies seek to minimize the complexity of the input, such that understanding flaws in programmatic logic is more straightforward for an operator.

D. End To End Methodology

Figure 3 provides an end-to-end overview of the ICSQuartz methodology, including program compilation, the fuzzing methodology, and the final security analysis.

Stage 1: Compilation. Initial programs are provided as plaintext ST files to our modified RuSTy compiler, along with the additional compatibility layer interfaces outlined in Table II. These interfaces provide the modified RuSTy compiler with the necessary interface definitions, such that the parameter and return types can all be validated at compile-time. The

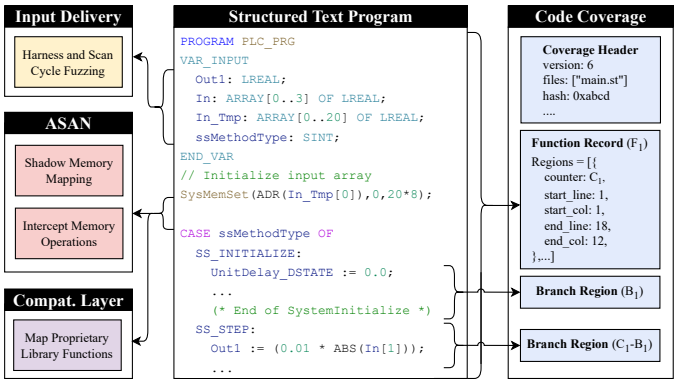

Fig. 4: **ICSQuartz Program Dissection.** Harnesses, memory sanitizers, compatibility layers, and coverage culminate to provide fine-grain security introspection and validation.

RuSTy compiler then performs an initial pass through the source code, parsing the syntax and constructing a program AST. Next, the ICSQuartz code coverage algorithm, included in 1, traverses the AST to construct instrumentation and inject metadata, including simple and expression-based counters for each execution branch. It also injects a coverage mapping header to the IR, which describes the implemented version of the LLVM coverage specification, paths to the source files, and hashes of function structures. Furthermore, the required ASAN instrumentation is introduced, ensuring extensive validation of program correctness pertaining to all memory-based operations. The resulting IR generated from RuSTy is then passed to `clang` [31], which provides the final step in the compilation, statically linking the necessary compatibility libraries invoked in the source input, along with the LLVM code coverage and ASAN runtime libraries.

Figure 4 provides a comprehensive diagram detailing how each component of the security instrumentation interacts simultaneously with a sample ST program, providing a higher degree of sophistication related to the analysis of program execution than previously possible.

Stage 2: Fuzzing. Once an Instrumented Control Application has been produced from Stage 1, ICSQuartz analyzes the input struct defined in the IR and leverages an LLM to reconstruct the variable memory layout in a C-struct. The output C is then inserted into a harness template, which allows for immediate fuzzing without an operator being required to create an initial harness from scratch. ICS-specific parameters, such as maximum scan cycle depth, can be configured to provide a more comprehensive analysis of the program's resilience. Other common fuzzing features, such as dictionaries and seeds, can also be introduced at this stage in order to improve fuzzing performance.

In the final stage, the program is compiled into an instrumented executable binary for in-process fuzzing by the ICSQuartz fuzzer, which enables scaling across multiple cores or machines over a network. As ICSQuartz performs a rigorous evaluation of the input ST program, a corpus of inputs is stored in memory and on disk, which contains inputs that each achieve unique coverage paths or cause a unique crash. Additional metadata pertaining to scan cycle exploration, stale scan cycles, and scan cycle mutations are recorded to enable comprehensive introspection into the fuzzing process.

Stage 3: Security Analysis. Standardization introduced by ICSQuartz considerably simplifies the fuzzing post-mortem for ICS binaries. The fuzz corpus provides a set of linearly independent inputs, each of which achieves additional code coverage. This corpus set can be used in conjunction with `llvm-cov` [24] to provide a line-by-line breakdown of fuzzer coverage and produce insights into portions of code that may require further evaluation. Memory-related crashes can be localized using ASAN, providing a comprehensive stack trace of the execution of the program leading up to the crash. These potential vulnerabilities can be further investigated, if necessary, using typical reverse engineering tooling, such as `gdb` or `Ghidra`.

V. Evaluation Setup

A. Structured Text Corpus

To exhaustively evaluate ICSQuartz against a diverse corpus of ST, we collected programs from a range of sources.

OSCAT Libraries: While significant portions of ST are proprietary and closed-source, the Open Source Community for Automation Technology (OSCAT) libraries stand out as a notable example of an open-source ICS software, which is distributed and widely used amongst a multitude of vendors, including Siemens, Moeller, Bosch, Beckhoff, Wago, and PC-Worx [25]. The OSCAT project distributes three main libraries: OSCAT Basic [25], OSCAT Building [32], and OSCAT Network [26]–comprised of 39,900, 6,422, and 23,786 lines of ST, respectively. OSCAT Basic includes a vast corpus of functions that operate on strings, implement different data structures, and perform calculations. OSCAT Building contains functions for building facilities, such as heating and air conditioning systems, and actuators [32]. OSCAT Network provides utilities for interacting with services over the network, including clients, cryptographic hashes, and binary encoders [26].

In total, we gather a total of 63 program benchmarks from OSCAT Basic and OSCAT Network, providing program templates and fuzzing harnesses to the community for future research. To our knowledge, this is the first instance of fuzzing an open-source ST library as a security research project.

ICSFuzz Benchmarks: The ICSFuzz project includes a set of 17 synthetic ST benchmarks [17], consisting of programs with four categories of vulnerabilities: Out-of-bounds write through `memcpy`, out-of-bounds write through `memset`, out of bounds read and write through `memmove`, and out of bounds reads and writes through arrays. These benchmarks increase in complexity, with benchmark numbering ranging from 1 to 13 and a total of 3,711 lines of ST.

ICSPatch Benchmarks: The ICSPatch project includes a set of 24 synthetic ST benchmarks [21], accumulating to 1,676 lines of ST. The ICSPatch programs are notable for two major reasons: 1) Benchmarks are modeled after real-world ICS applications, including an *Aircraft Flight Control*, and a

Control Application	Execution Speed (inputs/sec)			First Crash (seconds)			First Crash (inputs)		
	ICSQuartz (x64)	FieldFuzz (x64)	ICSFuzz (A8)	ICSQuartz (x64)	FieldFuzz (x64)	ICSFuzz (A8)	ICSQuartz (x64)	FieldFuzz (x64)	ICSFuzz (A8)
bf_mcpy_1	19649.9	593.0	70.9	0.0008	0.25	234	15.7	148	15270
bf_mcpy_6	1403.4	642.4	64.2	0.0140	1.43	188	19.6	898	12172
bf_mcpy_8	1355.6	645.6	66.1	0.0151	7.08	279	20.5	4566	18216
bf_mcpy_12	328.2	526.2	62.1	0.2630	1.95	426	86.3	999	26645
bf_mset_1	7526.0	560.6	64.6	0.0026	0.04	208	19.6	22	13441
bf_mset_3	9961.5	571.2	62.7	0.0020	0.03	174	19.5	17	10906
bf_mset_5	2227.1	503.2	68.8	0.0088	0.56	254	19.6	281	17554
bf_mmove_1	2626.4	660.2	64.6	0.0078	0.005	176	20.5	2	11245
bf_mmove_4	6674.9	578.2	63.1	0.0046	0.003	159	30.5	1	10070
bf_mmove_7	1924.3	573.0	66.3	0.0158	0.005	229	30.5	3	15317
bf_mmove_12	1932.5	508.2	64.5	0.0158	182.14	783	30.5	92456	50643
oob_1_arr_1	21747.2	598.8	71.9	0.0073	0.14	55	159.4	83	3880
oob_1_arr_6	53965.5	591.0	77.0	0.0068	1.39	103	367.1	821	8085
oob_1_arr_13	24039.7	507.0	75.2	0.0071	97.86	207	171.6	49165	27241
oob_2_arr_1	27174.2	520.8	73.5	0.0093	154.42	117	252.0	80080	8558
oob_2_arr_5	53055.0	520.4	71.1	0.0069	155.62	165	367.1	80662	22759
oob_2_arr_13	24999.5	502.2	71.0	0.0054	97.86	192	134.8	48694	13401
Average	15328.88	564.8	68.1	0.0231	41.22	232.29	103.81	21111.65	16788.41

TABLE III: **Performance of ICSQuartz vs. FieldFuzz [18] and ICSFuzz [17]**. Statistics as reported by respective work.

Desalination Plant. 2) These programs include vulnerabilities developed from MITRE's top applicable CWEs: Improper input validation (CWE-20), out-of-bounds write (CWE-787), out-of-bounds reads (CWE-125), and OS command injection (CWE-78) [33]. Similar to the OSCAT libraries, the ICSPatch programs ground the programs in real-world ICS applications, which incorporate the most common software vulnerabilities encountered by the industry.

Scan Cycle Benchmarks: In addition to the programs above, in this work, we introduce a new set of 12 synthetic benchmarks, which highlight the necessity for stateful scan cycle fuzzing, an execution model specific to the IEC 61131-3 programs with the potential to introduce vulnerabilities. These programs are based on the existing ICSPatch benchmarks, grounding them in real-world applications, and total 1,146 lines of ST. To the best of our knowledge, this type of program fuzzing has not been possible or explored by previous work. These benchmarks are also be open-sourced.

B. Fuzzing Infrastructure

Evaluations are performed on an Ubuntu 22.04.2 server, with 128 CPU cores and 2TB of allocated system memory. To validate ST vulnerabilities discovered by ICSQuartz, we use a WAGO PFC200 Controller (750-8216), running Codesys Runtime v3.5.16.10 and Codesys IDE v3.5 SP17, Patch 2. The OSCAT Basic used is the latest official Codesys port (3.3.4.0). In addition, we leverage the Codesys Control for Linux SL (3.5.16.10) to evaluate ICSFuzz [17] and FieldFuzz [18].

C. Fuzzing Practices

To standardize with industry best practices and conduct reproducible experiments [34, 35], we leverage `taskset` to conduct all fuzzing on the same CPU processor core for all experiments. In addition, we leverage `isolcpus` to disable the scheduling of tasks to our dedicated fuzzing processor core. All experiments are repeated a minimum of 10 times.

VI. ICSQuartz Evaluation

A. Performance in Existing ICS Benchmarks

Table III provides a comprehensive comparison of the fuzzing performance against the two state-of-the-art ICS fuzzing tools, ICSFuzz [17] and FieldFuzz [18], in 17 synthetic benchmarks. The comparison was conducted on three key metrics: execution speed (i.e., the number of program executions per second), time to find the first crash (in seconds), and the number of inputs required to reach the first crash. The results in Table III indicate that ICSQuartz outperforms both of the other ICS frameworks in all three metrics across the 4 categories of vulnerability benchmarks. The results show a statistically significant ($p<0.05$) improvement in all three metrics under a Mann-Whitney U-test, as recommended by the broader fuzzing community [34, 35].

As indicated in the first column, ICSQuartz exceeds the performance of FieldFuzz [18] and ICSFuzz [17] in terms of executions per second, surpassing them by factors of more than 27× and 225×, respectively. Improvements in executions per second come from a variety of factors: 1) Improved input delivery, where program input is passed through system memory rather than with the added overhead of a network, or the inconsistency of a runtime process, allowing ICSQuartz to provide higher consistency and lower latency than previous tools. 2) Vendor and platform independence allows programs to run as standard ELF executables on larger systems without vendor runtimes, which easily outperforms system resources available in PLCs.

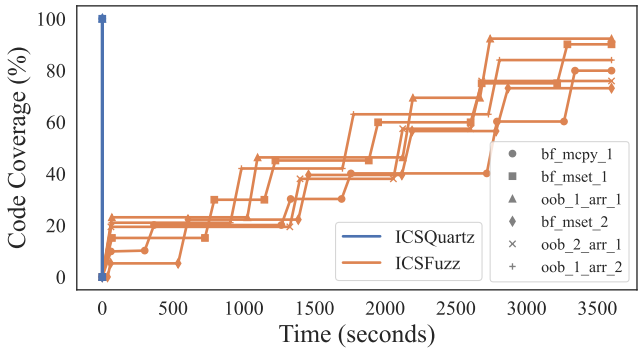

Fig. 5: **Coverage Performance in ICSFuzz Benchmarks.** Blue indicates ICSQuartz. Orange indicates ICSFuzz [17].

Moreover, investigating inputs to the first crash, ICSQuartz outperforms FieldFuzz [18] and ICSFuzz [17] in almost all benchmarks and by an average factor of 203× and 161×, respectively. Considering the inputs to the initial crash is essential when evaluating the performance of fuzzers: Although enhancements in sheer execution speed are beneficial, the number of inputs to the first crash demonstrates the overall performance of the input mutation strategy. Both improvements work in tandem: Each input is more effective, and more inputs are tested in the same duration. This combined improvement is seen in the time-to-first crash column, where ICSQuartz outperforms FieldFuzz [18] and ICSFuzz [17] by average factors of more than 1,784× and 10,055×.

Figure 5 represents a time graph of code coverage achieved, contrasting the performance of ICSQuartz with ICSFuzz [17] across multiple benchmarks. The visual differences highlight how the vendor and platform independence introduced by ICSQuartz improve substantially upon state-of-the-art fuzzers, with ICSQuartz immediately providing 100% code coverage for all benchmarks. Note that FieldFuzz [18] coverage data is not available.

B. Vulnerabilities Discovered

Table IV shows an excerpt of the fuzzing campaign conducted across the 63 OSCAT benchmarks, including the 25 programs that experienced a crash within one continuous hour of fuzzing. ICSFuzz [17] and FieldFuzz [18] were also evaluated across these benchmarks–with the exception of the OSCAT Network binaries, as this Codesys distribution of this library is not compatible with 64-bit architectures. Total executions for ICSFuzz and FieldFuzz are generally consistent due to reliance on the statically configured scan cycle speed, while ICSQuartz is able to perform executions without this dependency. ICSFuzz and FieldFuzz notably did not encounter any crashes, which we discuss further below.

1) OSCAT Vulnerabilities: ICSQuartz encountered a crash after 24 seconds of fuzzing the `MONTH_TO_STRING` function. Upon investigation, our team discovered that the function suffered from an out-of-bounds read vulnerability (CWE-125) due to improper input validation (CWE-20).

OSCAT Program	Total Executions		
	ICSQuartz	FieldFuzz	ICSFuzz
CHARNAME	123M	239k	102k
CLEAN	3.73M	239k	99.2k
DEL_CHARS	3.73M	239k	99.6k
DT_TO_STRF	129k	239k	102k
FIND_CHAR	*386k*	239k	102k
FIND_CTRL	*386k*	239k	99.1k
FINDB_NONUM	*436k*	239k	99.1k
FINDB_NUM	*436k*	239k	99.5k
FSTRING_TO_BYTE	817k	239k	99.3k
FSTRING_TO_DWORD	333k	239k	99.0k
IS_CC	3.73M	239k	99.2k
IS_NCC	3.73M	239k	99.3k
MIRROR	*436k*	239k	99.7k
MONTH_TO_STRING	**1.07B**	239k	102k
REAL_TO_STRF	399M	239k	102k
REPLACE_ALL	23.7M	239k	100k
REPLACE_CHARS	23.7M	239k	99.2k
TRIM	436k	240k	100k
TRIM1	436k	239k	99.3k
TRIME	436k	239k	100k
UPPER_CASE	436k	239k	99.7k
WEEKDAY_TO_STRING	966M	240k	102k
BASE64_ENCODE_STR	17.9k	N/A	N/A
XML_READER	*644k*	N/A	N/A

TABLE IV: **OSCAT Fuzzing Campaign. Bold** indicates a vendor-agnostic CVE issued (discovered only by ICSQuartz). *Italics* indicate a RuSTy-specific compiler vulnerability.

`MONTH_TO_STRING` accepts three input parameters: `MTH` (16-bit integer, as a numerical month), `LANG` (16-bit integer, as a language index: 1=English, 2=German, 3=French), and `LX` (16-bit integer, output length configuration, i.e. 0="January" (full month name), 3="Jan"), and returns a 10-byte string with the month name as a string. Although bound checks ensure that `MTH` and `LX` are in a valid range, and `LANG` is below the maximum allowed value, no bound checks ensure that `LANG` is non-negative. Providing a negative `LANG` allows an attacker to read 10 bytes at a time from any address higher than the global `language.MONTHS` definition.

Due to the notably large 132-byte size of each language's month definitions, the minimum integer value -32,768 of `LANG` can be used to traverse up to 4,325,376 $(132 \times 32,768)$ bytes up the stack. In addition, the `MTH` parameter can be adjusted between the values of 1 and 12, where each increment to MTH adds 11 bytes to the target address, moving back down the stack. The two of these parameters can be used very effectively in conjunction to traverse most target memory locations, leveraging both large and small jumps.

This vulnerability is particularly interesting, as it does not affect a single vendor but instead affects all vendors that distribute OSCAT Basic. Three major vendors are officially supported, with specific distributions directly from OSCAT: Codesys, PCWorx, and Siemens [25]. Codesys also distributes an official port of OSCAT Basic, which we reproduced the vulnerability on a real PLC. Although the vulnerability did impact Codesys systems, neither ICSFuzz [17] nor FieldFuzz [18] detected this vulnerability. We propose that this is likely due to the reliance of segmentation faults to detect vulnerabilities

	Fuzzer	Aircraft Oobr	Aircraft Oobw 4	Aircraft Oobw 5	Anaerobic Oobr 1	Anaerobic Oobr 2	Anaerobic Oobw 1	Anaerobic Oobw 2	Anaerobic Oobw 3	Chemical Oobr 1	Chemical Oobw 1	Smart Grid Oobr 1	Smart Grid Oobw 1
Crashes	ICSQuartz	10	8	10	10	10	5	10	10	10	10	10	10
	AFL++	1	0	0	0	1	0	0	0	0	0	3	1
	FieldFuzz	0	0	9	0	0	0	0	0	0	0	0	0
	ICSFuzz	0	0	0	0	0	0	0	0	0	0	0	0
Time (s)	ICSQuartz	12.4	56.9	0.8	1.5	0.1	30.7	3.1	0.1	0.2	0.2	0.1	0.5
	AFL++	78.8	-	-	-	0.8	-	-	-	-	-	2.4	0.9
	FieldFuzz	-	-	42.5	-	-	-	-	-	-	-	-	-
	ICSFuzz	-	-	-	-	-	-	-	-	-	-	-	-
Execs.	ICSQuartz	5.6M	14M	95k	560k	1.0k	14M	480k	240	1.6k	6.1k	2.4k	100k
	AFL++	32k	-	-	-	1.3k	-	-	-	-	-	1.6k	2.1k
	FieldFuzz	-	-	2.7k	-	-	-	-	-	-	-	-	-
	ICSFuzz	-	-	-	-	-	-	-	-	-	-	-	-

TABLE V: **Scan Cycle Benchmark Evaluation (10 Trials).** ICSQuartz is evaluated against state-of-the-art fuzzers. Existing mutation strategies are unable to uncover scan cycle vulnerabilities in 7 out of 12 benchmarks.

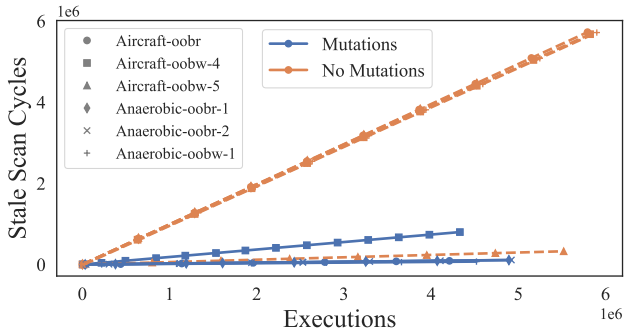

Fig. 6: **Mutations Curbing Stale Scan Cycles.** The two proposed mutation strategies effectively curb stale scan cycles.

rather than a more precise address sanitizer [30].

To responsibly disclose the vulnerability, our team submitted the vulnerability to the CERT Coordination Center (CERT / CC) on 2024-04-10. After corresponding with a security professional from Codesys and notifying the OSCAT developers, the vulnerability was assigned as CVE-2024-6876 and patched on 2024-10-09.

2) RuSTy Compiler Bug: Throughout the fuzzing campaign, a recurring bug in the RuSTy compiler was discovered, which introduces vulnerabilities into ST programs: Loops with negative increments were found to execute once without properly performing the condition check. Additional details on this vulnerability are described in Appendix B.

C. Scan Cycle Fuzzing Technique

In this section, we compare the scan cycle-aware mutation strategies of ICSQuartz with three fuzzers: **1)** AFL++ [28]

	ICSQuartz Scan Cycle Mutations	Aircraft Oobr	Aircraft Oobw 4	Aircraft Oobw 5	Anaerobic Oobr 1	Anaerobic Oobr 2	Anaerobic Oobw 1	Anaerobic Oobw 2	Anaerobic Oobw 3	Chemical Oobr 1	Chemical Oobw 1	Smart Grid Oobr 1	Smart Grid Oobw 1
Total Crashes	●	10	8	10	10	10	5	10	10	10	10	10	10
	○	0	0	0	0	1	0	0	0	0	0	0	0
Stale Cycles	●	1.9	20.9	31.3	1.0	1.5	1.3	40.5	30.2	3.1	3.1	6.6	1.9
	○	99.9	99.9	44.0	99.9	99.9	99.9	99.9	99.9	5.4	99.9	99.9	99.9

TABLE VI: **Evaluation of ICSQuartz Mutations (10 Trials).** ICSQuartz, without the proposed mutation strategies, is unable to reliably or efficiently detect scan cycle vulnerabilities.

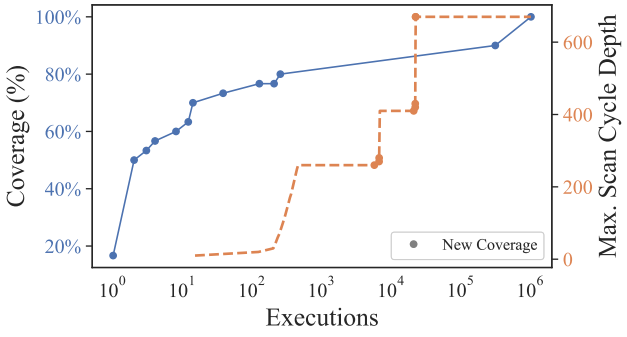

Fig. 7: **Coverage vs. Scan Cycle Depth (Aircraft-oobw-4).** Scan cycle programs require many scan cycle executions to effectively exhaust control flow paths and reach full coverage.

to represent traditional IT fuzzers. **2)** FieldFuzz [18], and **3)** ICSFuzz [17] to represent state-of-the-art ICS fuzzers.

These four fuzzers are each evaluated in Table V across twelve synthetic scan cycle benchmarks–each over ten independent trials. The featured metrics denote the number of trials the respective fuzzer discovered the vulnerability, the time to discover, and the number of executions required. Lastly, the last column includes the averaged stale scan cycles encountered overall–reported only for ICSQuartz and the ICSQuartz variant, as the remaining fuzzers are not scan cycle-aware. Table V demonstrates that all compared fuzzers *are unable to discover the vulnerability in 7 out of 12 benchmarks* evaluated. In contrast, the ICSQuartz mutation strategy discovers all bugs and demonstrates the vital role of the mutation strategies by its stark contrast with the ICSQuartz variant.

We then investigate the efficacy of the proposed mutation strategies in Table VI. In these experiments, we compare ICSQuartz against a naive variant of ICSQuartz that excludes the proposed mutations and instead only implements the standard AFL++ mutations [28, 27]. Further, it statically resets the program state after a constant number of scan cycles to introduce a naive approach to scan cycle awareness. These empirically demonstrate the impact of the proposed mutation strategies in

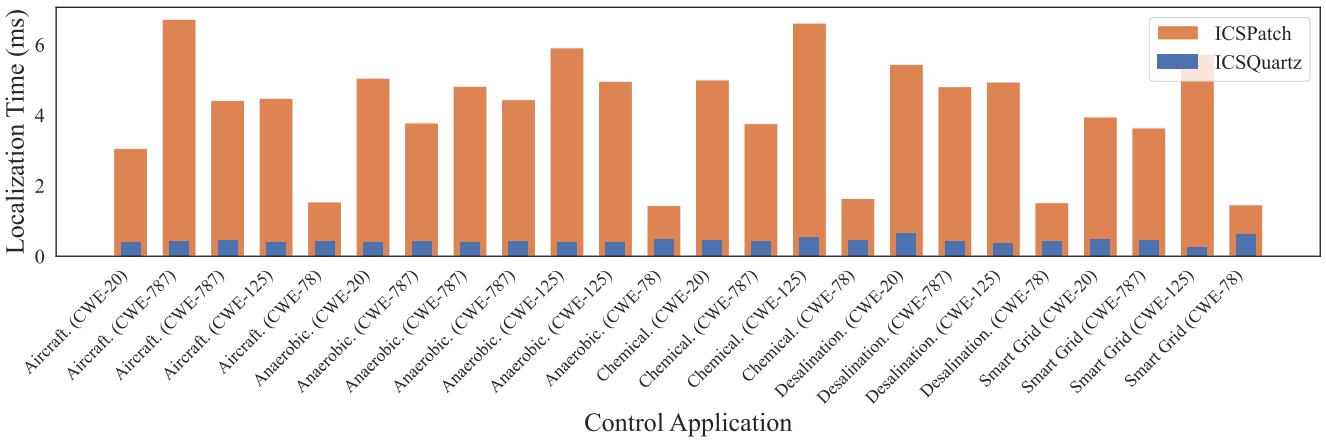

Fig. 8: **ICSQuartz vs. ICSPatch [21] in Vulnerability Localization.** ICSQuartz, due to a significant decrease in instrumentation overhead and white-box access, is able to localize vulnerabilities faster than ICSPatch across all CWE-based benchmarks.

scan cycle-aware benchmarks, where fewer wasted executions lead to more effective vulnerability detection.

Figure 6 details the fuzzing process in more detail, plotting the number of stale scan cycles encountered against the total number of executions. The standard AFL++ [28] algorithm is susceptible to stale scan cycles, where almost all executions performed are unnecessary. In contrast, the adaptive scan cycle algorithms in ICSQuartz perform well in reliably and consistently avoiding these paths. This figure reinforces the importance of scan cycle-specific components, which complement existing state-of-the-art fuzzing strategies.

Introspecting deeper into the scan cycle behavior introduced by ICS programs, we investigate the relationship between coverage and the maximum scan cycle depth explored by ICSQuartz. Figure 7 depicts a performance analysis on the Aircraft Flight Control benchmark (Aircraft Oobw 4) introduced in Table V, which includes a program input size of 32 bytes. We chose this specific benchmark as it features a diverse range of control flow branching and paths for scan cycle exploration. This figure accentuates the crucial component scan cycles introduced to ICS programs and robust security evaluations of ICS software. As stateful executions are rigorously evaluated, novel and unexpected execution paths are uncovered.

D. Vulnerability Localization

To understand how effectively ICSQuartz can detect some of the most prevalent vulnerabilities in software, we evaluate against the 24 synthetic benchmarks provided by ICSPatch [21]. Figure 8 visualizes the substantial improvement in vulnerability location provided by ICSQuartz, outperforming ICSPatch in all benchmarks. On average, ICSQuartz localizes vulnerabilities 9× faster than the state-of-the-art. Table VII provides a qualitative analysis of the vulnerability localization process in a single sample: Aircraft Flight Control (CWE-20).

These results demonstrate that when white-box access is available, the vulnerability localization process is 185× faster on the ICSQuartz platform and does not require access to a PLC testbed for hexdump extraction. Though ICSQuartz does not compete directly with ICSPatch, we suggest that ICSQuartz vulnerability localization may be a suitable addition to ICSPatch when possible, enabling improved performance while being vendor-agnostic.

	Phase	Prep. (s)	Vulnerability Localization (s)		
ICSQuartz	**Steps**	Compile ST	ST Execution		ASAN
	Device	LLVM Target			
	Time	3.624	0.007		0.638
ICSPatch [21]	**Steps**	Extract Hexdump	Load Hexdumps	Program Execution	DDG Traversal
	Device	PLC Testbed	ICSPatch Host (angr)		
	Time	733.11	52.02	4.73	0.003

TABLE VII: **Comparison of Vulnerability Localization.** ICSQuartz improves upon the vulnerability localization process compared to ICSPatch [21] in speed and usability.

VII. Discussion

A. ICSQuartz Limitations

In this work, ICSQuartz empirically demonstrated statistically significant improvements upon state-of-the-art ICS fuzzers in both performance and scan cycle fuzzing. Nevertheless, there are cases where a hybrid-fuzzing approach may be advantageous in order to evaluate a broader threat model. To that end, operators may choose ICSQuartz to rigorously evaluate stateful PLC-logic and libraries under a white-box setting, while concurrently leveraging FieldFuzz [18] or ICS-

Fuzz [17] to proactively detect potential weaknesses in the PLC runtime or physical hardware.

RuSTy Compiler: Although the use of a relatively new open-source compiler introduces the potential for compilation bugs or discrepancies between fuzzing targets and PLC binaries, we contend that the significant benefits of white-box access far outweigh the potential downsides.

Furthermore, all potential downsides of the RuSTy compiler can be further alleviated through a rigorous cross-validation process: Leveraging traditional ST compilers such as the Codesys IDE [20] or SIEMENS TIA Portal [36] and reproducing vulnerabilities on a real-world PLC Testbed, as we demonstrated with the OSCAT vulnerability we uncovered and reported to the vendor, and it was reproducible when using their own proprietary compiler.

Precompiled Libraries Fuzzing: Fuzzing proprietary libraries can require more sophisticated techniques: FieldFuzz [18] can fuzz network-reachable runtime components, while ICS-Fuzz [17] is capable of fuzzing proprietary function interfaces directly whilst running on a virtual or physical Codesys. SP-Fuzz [37] further tackles these challenges–introducing dynamic taint analysis to extract contextual information about the target from the PLC runtime to dynamically generate a fuzzing harness. As ICSQuartz is a white-box fuzzer, it is unable to perform security evaluation of proprietary precompiled libraries that do not include source code. These complementary relationships underscore the vital role a hybrid approach to ICS fuzzing can play in mitigating vulnerabilities.

Automation of Cross-Validation: While our framework is entirely automated, reproducing crashes and cross-validating potential vulnerabilities on a physical PLC testbed remains challenging to fully automate: The process entails manual hardware configurations on the PLC, compilation of the ST using the proprietary vendor IDEs, and analysis of program execution as inputs are executed. We leave the automation of PLC-based cross-validation as a problem to be tackled in future work. Additionally, multi-vendor cross-validation could further enable a rigorous evaluation and differential comparison across multiple platforms and hardware devices.

The specific impact of vulnerabilities will depend on context, including memory layouts and security features (e.g. non-executable stack), which vary between vendors, PLC firmware, and hardware. Case in point, our RuSTy bug discovery is highly-critical for programs compiled with the RuSTy compiler, but does not reproduce with other PLC compilers. However, all OSCAT vulnerabilities discovered have been successfully reproduced using the proprietary Codesys compiler.

B. Directions for Future Research

ICSQuartz serves as an initial bridge from robust and long-standing traditional IT fuzzing tools [28, 38, 39] into the previously vendor-locked ICS fuzzer research [17, 18]. In that respect, there exists significant room for future contributions that continue to build on existing work while introducing unique ICS-specific novelty that must be accounted for.

Scan Cycle Fuzzing: While ICSQuartz introduces several key fuzzing components and strategies that uncover scan cycle-specific vulnerabilities, future research can continue to build and reuse these components. The introduction of scan cycles into the fuzzing process opens the door to considerable problems of state explosion. While ICSQuartz partially tames this state explosion challenge through the detection of stale scan cycles, future work could investigate how states could be stored and minimized natively in fuzzer test cases.

ML-Guided Fuzzing: Machine learning and deep learning are active areas for exploration, improving upon classical fuzzing techniques in interesting ways. The unique capabilities of LLMs have become a recent valuable addition to the field of software fuzzing and vulnerability discovery [40]: TitanFuzz [41], ChatAFL [42], LLMIF [43], and Fuzz4all [44] fine-tune LLMs to generate input for fuzzing IoT protocols, and system under test (SUTs). DeepGo [45] leverages reinforcement learning, proposing the first path transition model which models Directed Greybox Fuzzing. These novel techniques may be particularly useful in the ICS domain.

Emerging Fuzzing Techniques: Fuzzing performance may be further enhanced through other techniques, such as symbolic execution [46, 47], which has proven to be particularly effective in state-dependent branch programs. Research in this direction could complement the scan cycle novelty of ST programs. KLEE [48], a symbolic execution tool, is capable of automatically generating tests that achieve impressive coverage and high accuracy in bug detection. Ferry [49] introduced program-state-aware symbolic execution to efficiently explore the state-dependent branches. Finally, SYMSAN [50] improves upon concolic execution [51], and demonstrated its performance in exploring paths that are guarded by complex and tight branch predicates compared to random mutation-based fuzzing. Other work–such as Sizzler [19]–introduce Generative Adversarial Networks (GANs) to further optimize mutation strategies in fuzzing ladder diagrams.

VIII. Conclusion

This paper presents ICSQuartz, an ICS-specific fuzzer that improves upon state-of-the-art fuzzing components and follows a vendor-agnostic methodology to provide platform independence. Leveraging an IEC 61131-3 Structured Text open-source compiler, we instrument and comprehensively fuzz 116 ICS programs both with and without scan cycle considerations. This approach culminates in the first large-scale ICS code fuzzing and vulnerability analysis on real-world open-source libraries. Using ICSQuartz, we report multiple vulnerability disclosures and submit important bug fixes to the RuSTy compiler for future research to leverage. In addition, we accelerate fuzzing performance by more than an order of magnitude compared to state-of-the-art ICS fuzzing tools–evaluating directly against prior work. Finally, we present scan cycle-aware mutation strategies that uncover vulnerabilities not discovered by existing strategies. With ICSQuartz, we provide the research community with a scalable open-source fuzzer to facilitate research in ICS program fuzzing.

References

[1] K. H. John and M. Tiegelkamp, *IEC 61131-3: Programming industrial automation systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[2] L. Shi, S. Li, and X. Fu, "The Fourth Industrial Revolution, Technological Innovation and Firm Wages: Firm-level Evidence from OECD Economies," *Revue d'économie industrielle*, no. 169, pp. 89–125, Sep. 2020.

[3] C. Doumanidis, Y. Xie, P. H. Rajput, R. Pickren, B. Sahin, S. Zonouz, and M. Maniatakos, "Dissecting the industrial control systems software supply chain," *IEEE Security & Privacy*, vol. 21, no. 4, pp. 39–50, 2023.

[4] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[5] M. Intelligence, "IRONGATE ICS Malware: Nothing to See Here...Masking Malicious Activity on SCADA Systems," 2016, accessed: September 2023. [Online]. Available: https://www.mandiant.com/resources/blog/irongate-ics-malware

[6] U. C. Guard, "Cyberattack Impacts MTSA Facility Operations," 2019, accessed: September 2023. [Online]. Available: https://www.dco.uscg.mil/Portals/9/DCO%20Documents/5p/MSIB/2019/MSIB_10_19.pdf

[7] "Enisa threat landscape 2022," European Union Agency for Cybersecurity (ENISA), Tech. Rep., 2022. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022

[8] "Enisa threat landscape 2023," European Union Agency for Cybersecurity (ENISA), Tech. Rep., 2023. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2023

[9] B. Fernández Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, "Applying model checking to industrial-sized plc programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[10] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, p. 326–336.

[11] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao, "Towards Automated Safety Vetting of PLC Code in Real-World Plants," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 522–538.

[12] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of plc software," *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[13] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal Verification of Safety PLC Based Control Software," in *Integrated Formal Methods*, E. Ábrahám and M. Huisman, Eds. Cham: Springer International Publishing, 2016, vol. 9681, pp. 508–522.

[14] D. Darvas, B. Fernández Adiego, A. Vörös, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez, "Formal verification of complex properties on PLC programs," in *Formal Techniques for Distributed Objects, Components, and Systems*, E. Ábrahám and C. Palamidessi, Eds. Springer, 2014, pp. 284–299.

[15] A. Keliris and M. Maniatakos, "ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[16] Beckhoff, "Beckhoff twincat machine learning," 2019. [Online]. Available: https://www.beckhoff.com/en-en/products/automation/twincat-3-machine-learning/

[17] D. Tychalas, et al, "ICSFuzz: Manipulating I/Os and repurposing binary code to enable instrumented fuzzing in ICS control applications," in *30th USENIX Security Symposium*, 2021, pp. 2847–2862.

[18] A. Bytes, P. H. N. Rajput, C. Doumanidis, M. Maniatakos, J. Zhou, and N. O. Tippenhauer, "FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM, 2023, pp. 499–512.

[19] K. Feng, M. M. Cook, and A. K. Marnerides, "Sizzler: Sequential fuzzing in ladder diagrams for vulnerability detection and discovery in programmable logic controllers," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 1660–1671, 2024.

[20] C. Group, "Codesys inside directory," 2023, accessed October 2023. [Online]. Available: https://www.codesys.com/the-system/codesys-inside.html

[21] P. H. N. Rajput, C. Doumanidis, and M. Maniatakos, "Icspatch: automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023.

[22] C. Group, "Codesys online help: Operator new," 2024. [Online]. Available: https://help.codesys.com/api-content/2/codesys/3.5.12.0/en/_cds_operator_new/

[23] P.-L. Group, "Rusty compiler," https://plc-lang.github.io/rusty/, 2024, accessed: January 2024.

[24] L. D. Group, "Llvm platform," https://llvm.org/, 2024, accessed: January 2024.

[25] OSCAT, "Oscat basic library," http://www.oscat.de/de/component/jdownloads/category/2-oscat-basic.html, 2012, accessed: January 2024.

[26] ——, "Oscat network library," http://www.oscat.de/de/component/jdownloads/category/3-oscat-network.html, 2015, accessed: January 2024.

[27] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1051–1065.

[28] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++:

combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020.

[29] The Rust Community, "Inkwell: It's a New Kind of Wrapper for LLVM," https://github.com/TheDan64/inkwell, 2021, accessed: April 2024.

[30] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proceedings of the USENIX Conference on Annual Technical Conference*, 2012, p. 28.

[31] L. D. Group, "Clang compiler," https://clang.llvm.org/, 2024, accessed: January 2024.

[32] OSCAT, "Oscat building library," http://www.oscat.de/de/component/jdownloads/category/5-oscat-building.html, 2015, accessed: January 2024.

[33] MITRE, "2021 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021, accessed: April 2024.

[34] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., 2018, pp. 2123–2138.

[35] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent Evaluation Practices for Fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1974–1993.

[36] S. D. I. Software, "Totally integrated automation (tia) portal," https://support.industry.siemens.com/cs/products?pnid=21813&cache=1598275693, Siemens 2024, accessed: July 2024.

[37] S. Jeon and J. Seo, "Sp-fuzz: Fuzzing soft PLC with semi-automated harness synthesis," in *Information Security Applications - 24th International Conference,*, H. Kim and J. M. Youn, Eds., vol. 14402. Springer, 2023, pp. 282–293.

[38] L. Project, "Libfuzzer," https://llvm.org/docs/LibFuzzer.html, 2024, accessed: April 2024.

[39] Google, "Honggfuzz," https://honggfuzz.dev/, 2022, accessed: April 2024.

[40] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li, and Q. Zhang, "When Fuzzing Meets LLMs: Challenges and Opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 492–496.

[41] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.

[42] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model guided Protocol Fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2024.

[43] J. Wang, L. Yu, and X. Luo, "LLMIF: Augmented Large Language Model for Fuzzing IoT Devices," in *IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 881–896.

[44] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal Fuzzing with Large Language Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[45] P. Lin, P. Wang, X. Zhou, W. Xie, G. Zhang, and K. Lu, "DeepGo: Predictive Directed Greybox Fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2024.

[46] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*. ACM, 1975, pp. 234–245.

[47] W. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 266–278, 1977.

[48] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, R. Draves and R. van Renesse, Eds., 2008, pp. 209–224.

[49] S. Zhou, Z. Yang, D. Qiao, P. Liu, M. Yang, Z. Wang, and C. Wu, "Ferry: State-Aware symbolic execution for exploring State-Dependent program paths," in *31st USENIX Security Symposium*, 2022, pp. 4365–4382.

[50] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis," in *31st USENIX Security Symposium*, 2022, pp. 2531–2548.

[51] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 122–131.

[52] MITRE, "CVE-2008-1367: Memory Corruption Vulnerability in gcc 4.3.x," https://nvd.nist.gov/vuln/detail/CVE-2008-1367, 2008, published 2008-03-17, Updated 2017-09-29.

APPENDIX A
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The latest source is available at: github.com/momalab/ICSQuartz (10.5281/zenodo.14249993).

2) Hardware dependencies: A commodity computer.

3) Software dependencies: Linux system (validated on Ubuntu 22.04), Git, Docker (validated with 27.3.1), Python (requires 3.10 or higher), Python pip (`python3-pip`), and Python virtual environment (`python3-venv`).

4) Benchmarks: Included under `./benchmarks`.

B. Artifact Installation & Configuration

To validate the major claims of this work:

1) Install all dependencies.
2) Add `$USER` to the `docker` group.
3) Clone and enter the GitHub repository.
4) Install the Python libraries in `requirements.txt`.
5) Disable ASLR to run ICSFuzz [17] and FieldFuzz [18].
6) Calibrate the CODESYS virtual PLC for ICSFuzz and FieldFuzz: `./scripts/calibrate-codesys.sh`.

C. Major Claims

- (C1): ICSQUARTZ outperforms prior work by more than an order of magnitude in execution speed (Table III).
- (C2): ICSQUARTZ is the first, to our knowledge, to discover a real-word ST vulnerability (Table IV).
- (C3): ICSQUARTZ introduces a novel scan-cycle fuzzing technique with two new mutation strategies (Table V).

D. Evaluation

The `run_experiment.py` script orchestrates the benchmark build process and concurrent fuzzing with parameters:

- `--fuzz-time`: seconds to fuzz each benchmark.
- `--fuzz-trials`: times to repeat each experiment.
- `--cpus`: cores available for fuzzing (i.e. `1-8`). **Note:** ICSFuzz and FieldFuzz can experience issues when running in parallel across many cores.
- `--experiment`: table to reproduce (i.e. `table_3`).

Invoking an experiment script will automatically:

1) Compile the program source into an instrumented binary.
2) Build fuzzing targets using the respective fuzzer.
3) Execute fuzzing in batches of size: `|cpus|`.
4) Collect and aggregate statistics into: `results/`.

The time required for the build stages will vary, and may take significantly longer for the first experiment as dependencies are downloaded and built in the containers. The time required for the actual fuzzing (after building) should be approximately:

$$\left\lceil \frac{\texttt{fuzz-time} \times \texttt{fuzz-trials} \times |\texttt{benchmarks}|}{|\texttt{cpus}|} \right\rceil$$

E. Run All Experiments (E1-E5)

You may execute all 5 experiments sequentially by running: `./evaluate-all.sh`. The script will output results in `./all-results.txt`.

1) Experiment (E1): [Performance] [Table III] [10 human-minutes + 1.5 compute-hour]: Here we compare with state-of-the-art ICS fuzzers.

[How to]
```
./run_experiment.py \
--fuzz-time 565 --fuzz-trials 3 \
--cpus 1-8 --experiment table_3
```

[Results] Metrics to compare are `execs_per_sec`, `first_crash_time` and `first_crash_executions`. While executions per second will vary significantly depending on the hardware, the inputs to first crash should not.

2) Experiment (E2): [Fuzzing Campaign] [Table IV] [10 human-minutes + 1.5 compute-hour]: Here we reproduce the fuzzing campaign across the OSCAT Basic library using a subset of 18 benchmarks.

[How to]
```
./run_experiment.py \
--fuzz-time 170 --fuzz-trials 3 \
--cpus 1-8 --experiment table_7
```

[Results] Compare total executions between ICSQuartz, FieldFuzz, and ICSFuzz. As ICSQuartz is not tied to scan cycles, total executions should outperform significantly.

3) Experiment (E3): [CVE] [10 human-minutes + 0.2 compute-hour]: Here we reproduce the disclosed CVE.

[How to]
```
./run_experiment.py \
--fuzz-time 60 --fuzz-trials 1 \
--cpus 1-8 --experiment cve
```

[Results] This demonstrates the additional precision of ICSQuartz for detecting memory vulnerabilities. A crash should be quickly detected by ICSQuartz, but will not be detected by FieldFuzz and ICSFuzz.

4) Experiment (E4): [Scan Cycle Fuzzing] [Table V] [10 human-minutes + 0.2 compute-hour]: Here we reproduce the ICSQuartz scan cycle fuzzing campaign across 12 benchmarks and compare it with ICSFuzz and FieldFuzz.

[How to]
```
./run_experiment.py \
--fuzz-time 80 --fuzz-trials 3 \
--cpus 1-8 --experiment table_4
```

[Results] This demonstrates how ICSQuartz locates vulnerabilities not reliably detected by prior work.

5) Experiment (E5): [Scan Cycle Fuzzing] [Table VI] [10 human-minutes + 0.2 compute-hour]: Here we reproduce the ICSQuartz mutation strategy evaluation.

[How to]
```
./run_experiment.py \
--fuzz-time 80 --fuzz-trials 3 \
--cpus 1-8 --experiment table_5
```

[Results] The `state_resets` metric indicates the scan cycle mutation algorithm interventions to reset stale execution paths. The higher number of `first_crash_executions` in these benchmarks reflects the stateful complexity introduced by ST programs tracking residual states.

F. Note on Artifact Revision

While this work was revised from its initial submission, the AEC evaluated the revised version of this artifact.

APPENDIX B
RUSTY VULNERABILITY

This ST implementation error in RuSTy introduces vulnerabilities to at least 6 OSCAT Basic [25] functions and likely introduces vulnerabilities in other programs and libraries not evaluated in our fuzzing campaign. This bug parallels a high-severity vulnerability documented over 10 years ago in `gcc`, where the generation of a bad instruction led to memory corruption vulnerabilities introduced to compiled programs [52].

We then identified the root cause of this defect in the IR code generated by RuSTy: A signed less than or equal (`sle`) instruction that should instead be a signed greater than or equal to (`sge`) instruction to properly perform bound checks, in the case where the loop step size is negative. Listing 2 provides an example of this bug in ST, where the loop will run exactly once before terminating, along with the corresponding condition check as LLVM IR.

The security impact of this is context-dependent to the application running and the specific memory layout. In severe cases, this bug introduces vulnerabilities that enable out-of-bounds reads (CWE-125), out-of-bounds writes (CWE-787), or OS-command injection (CWE-78). We disclosed this implementation bug to the RuSTy compiler team on 2024-04-24 through a GitHub security advisory report, and a patch was issued on 2024-06-27.

We discovered several shortcomings in the RuSTy's implementation of the IEC 61131-3 standard library, which account for crashes in the fuzzing campaign. Implemented in the Rust programming language, many of the string functions, such as `LEN`, expect inputs to be valid ASCII characters. In cases where non-ASCII characters were provided, the implementation would most often call a Rust panic, successfully mitigating memory vulnerabilities, through crashing the program.

Source Code:

```
FOR a := 0 TO 10 BY -1 DO;
    printf('Hello %d!$N', a);
END_FOR;
```

LLVM IR:

```
condition_check:                  ; preds = %increment, %
    entry
  %load_a = load i16, i16* %a, align 2, !dbg !22
  %load_a1 = load i16, i16* %a, align 2, !dbg !22
- %tmpVar = icmp sle i16 %load_a1, 10, !dbg !22
+ %tmpVar = icmp sge i16 %load_a1, 10, !dbg !22
...
```

Listing 2: **Instance of RuSTy Bug and Vulnerable Code.** The loop shown in the code shown will execute once before terminating. The execution is due to the `sle` instruction, which should instead be a `sge`, as shown in the IR.

APPENDIX C
LLM HARNESS

Listing 3 presents the LLM prompt used to produce a ST entrypoint program, while Listing 4 presents a prompt to construct a C-style harness for fuzzing.

System: You are a helpful assistant who provides C-style structures when prompted with LLVM IR. Please refer to the following example in your responses:

User: Complete a C++ struct for this LLVM type. Assume stdint.h is already imported:

```
%PLC_PRG = type { i32, i32, i32, [11 x i8] }
```

Assistant:

```
struct PLC_PRG_struct {
    int32_t field1;
    int32_t field2;
    int32_t field3;
    int8_t array1[11];
};
```

Listing 3: **Prompt for LLM-Generated Harness.** The above prompt was utilized to dynamically generate C-style structures using the LLVM IR from the respective compiler. The output struct is then used within a standard harness template–automating the harness generation entirely.

System: Provided a function documentation in IEC 61131-3 Structured Text, please return a PROGRAM (PLC_PRG), including inputs to invoke the function.

Please refer to the following example for future responses:

User: 13.59. MONTH_TO_STRING
Type Function: STRING (10)
Input MTH: INT (Month 1..12)
LANG: INT (Language 0 = Default)
LX: INT (length of string)
Output STRING (10) (output value)
MONTH_TO_STRING convert a month number to its equivalent string...

Assistant:

```
PROGRAM PLC_PRG
VAR_INPUT
    in_MTH: INT;
    in_LANG: INT;
    in_LX: INT;
END_VAR

MONTH_TO_STRING(in_MTH, in_LANG, in_LX);

END_PROGRAM
```

Listing 4: **Prompt to Construct a PLC Entrypoint.** The example documentation is for the `MONTH_TO_STRING` function provided by OSCAT Basic [25].

APPENDIX D
SAMPLE STRUCTURED TEXT PROGRAM

Listing 5 presents a real-world ST sample extracted from the OSCAT basic library.

```
FUNCTION REPLACE_UML : STRING[STRING_LENGTH]
VAR_INPUT
    str : STRING[STRING_LENGTH];
END_VAR
VAR
    L : INT;
    pt : REF_TO BYTE;
    pto : REF_TO BYTE;
    ptm : REF_TO BYTE;
    pt1, pt2 : REF_TO BYTE;
    su : STRING[2];
    pos : INT;
END_VAR
VAR_TEMP
    ptot,ptmt: LWORD;
END_VAR

PT := REF(str);
pto := REF(REPLACE_UML);
ptm := pto + INT_TO_DWORD(string_length);
pt1 := REF(su);
pt2 := pt1 + 1;
L := LEN(str);
WHILE pos < L AND pos < string_length DO
    IF pt^ < 127 THEN
        (* no uml character simlply copy the
            character*)
        pto^ := pt^;
    ELSE
        (* convert the uml character *)
        su := TO_UML(pt^);
        (* we must make sure pointer are not out of
            range *)
        pto^ := pt1^;
        ptot := pto;
        ptmt := ptm;
        IF ptot < ptmt AND pt2^ > 0 THEN
            pto := pto + 1;
            pto^ := pt2^;
        END_IF;
    END_IF;
    (* increment pointers *)
    pt := pt + 1;
    pto := pto + 1;
    pos := pos + 1;
END_WHILE;

(* properly close the output string *)
pto^ := 0;

END_FUNCTION
```

Listing 5: **Sample of ST Program.** This program was extracted from the OSCAT Basic [25] library.