An LLM-Driven Fuzzing Framework for Detecting Logic Instruction Bugs in PLCs

Jiaxing Cheng[†]
Institute of Information Engineering, CAS; SCS, UCAS
Beijing, China
chengjiaxing@iie.ac.cn

Ming Zhou[†]
SCS, Nanjing University of Science and Technology
Nanjing, Jiangsu, China
mingzhou@njust.edu.cn

Haining Wang
ECE
Virginia Tech
Arlington, VA, USA
hnw@vt.edu

Xin Chen[*]
Institute of Information Engineering, CAS; SCS, UCAS
Beijing, China
chenxin1990@iie.ac.cn

Yuncheng Wang
Institute of Information Engineering CAS; SCS, UCAS
Beijing, China
wangyuncheng@iie.ac.cn

Yibo Qu
Institute of Information Engineering CAS; SCS, UCAS
Beijing, China
quyibo@iie.ac.cn

Limin Sun[*]
Institute of Information Engineering CAS; SCS, UCAS
Beijing, China
sunlimin@iie.ac.cn

***Abstract*—Programmable Logic Controllers (PLCs) automate industrial operations using vendor-supplied logic instruction libraries compiled into device firmware. These libraries may contain security flaws that, when exploited through physical control routines, network-facing services, or PLC runtime subsystems, may lead to privilege violations, memory corruption, or data leakage. This paper presents LogicFuzz, the first fuzzing framework designed specifically to target logic instructions in PLC firmware. LogicFuzz constructs a semantic dependency graph (SDG) that captures both operational semantics and inter-instruction dependencies in PLC code. Leveraging the SDG together with an enable-signal mechanism, LogicFuzz automatically synthesizes instruction-tailored seed programs, significantly reducing manual effort and enabling controlled, resettable fuzzing on real PLC hardware. To uncover bugs conditioned on control-flow triggers (i.e., invocation patterns), LogicFuzz mutates the SDG to diversify instruction-invocation contexts. To expose data-triggered faults, it performs coverage-guided parameter mutation under valid semantic constraints. In addition, LogicFuzz integrates a multi-source oracle that monitors runtime logs, status LEDs, and communication states to detect instruction-level failures during fuzzing. We evaluate LogicFuzz on six production PLCs from three major vendors and uncover 19 instruction-level bugs, including four previously unknown vulnerabilities.**

I. Introduction

Programmable Logic Controllers (PLCs) underpin critical infrastructure, including power generation, water treatment, and industrial manufacturing. As societal reliance on these systems grows, faults in PLCs can disrupt essential services and, in severe cases, threaten national security and public safety [9] [36] [31] [21]. The Triton (TRISIS) attack [17] exemplifies these stakes: adversaries exploited zero-day vulnerabilities in Schneider Electric safety PLCs to tamper with emergency shutdown logic, nearly causing catastrophic damage at a Saudi petrochemical facility.

PLCs execute control programs built from logic instructions that drive physical actuation (e.g., motion control), manage communication interfaces (e.g., Modbus services), and coordinate device-internal subsystems (e.g., memory management). Vendors commonly package these instructions into proprietary firmware-embedded libraries. However, these libraries may contain defects—including unchecked inputs, memory-safety violations (e.g., buffer overflows, null pointer dereferences), and race conditions—introduced through implementation mistakes. When triggered by malformed invocation patterns or adversarial parameters, such bugs can corrupt memory or escalate privileges, endangering the industrial processes the PLC governs [13]. Compounding this risk, vendors frequently reuse instruction libraries across product lines, enabling a single flaw to propagate multiple PLC models.

We propose an automated framework for uncovering logic-instruction bugs in PLC firmware. Conventional static analysis techniques [12] [32] (e.g., taint analysis) are ill-suited for proprietary PLC images: firmware binaries are often stripped, use non-standard formats, and tightly integrate vendor-specific runtime components. As a result, static tools cannot reliably recover code boundaries, data layouts, or calling conventions—context that is essential to modeling instruction semantics. Without this context, they are also unable to infer the data- and control-flow dependencies required for instruction-level bug detection, rendering static approaches largely ineffective on closed PLC platforms. Given these constraints, we turn to fuzzing, which exercises logic instructions through vendor-supported runtime interfaces without requiring visibility into firmware internals. This choice introduces three key challenges that motivate our design.

[†] These authors contributed equally to this work.
[*] The corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241081
www.ndss-symposium.org

Challenge 1: Synthesizing semantics-aware seed programs. Fuzzing logic instruction requires test programs that adhere to usage constraints, hardware side effects (e.g., I/O, timers, watchdogs), and PLC scan-cycle semantics. These behaviors vary significantly across vendors and are poorly documented in closed-source platforms, making manual derivation slow and error-prone. The challenge is to accurately recover instruction-level usage semantics and automatically generate valid, controllable, and resettable seed programs at scale.

Challenge 2: Feedback-guided exploration without instrumentation. PLC firmware is proprietary, heterogeneous across architectures and RTOSes, and incompatible with conventional techniques such as instrumentation or binary rewriting—mechanisms that typically supply coverage or state feedback. In the absence of such signals, fuzzers struggle to guide execution toward high-value states and corner cases. The challenge is to extract meaningful, low-noise feedback from vendor-exposed interfaces and runtime artifacts, and to design mutation strategies that effectively leverage this feedback to produce high-quality inputs.

Challenge 3: Comprehensive anomaly detection beyond crashes. Logic instruction failures frequently manifest not as crashes but as scan-cycle stalls, I/O inconsistencies, timing violations, watchdog resets, or silent state divergence—behaviors that crash-only oracles systematically miss. Meanwhile, PLCs expose heterogeneous exception signals (e.g., runtime logs, status LEDs, and communication states) that are encoded in vendor-specific ways. The challenge is to build a vendor-agnostic monitoring oracle that fuses these signals to detect a broad spectrum of anomalies while minimizing false positives.

We present **LogicFuzz**, an automated fuzzing framework for PLC logic instructions. To the best of our knowledge, LogicFuzz is the first system to systematically target instruction-level bugs on real PLC hardware. LogicFuzz constructs a lightweight semantic dependency graph (SDG) and couples it with an enable-signal-guided prompting mechanism that steers a large language model (LLM) to generate test programs that are valid, controllable, and resettable while adhering to hardware constraints and instruction semantics. To drive exploration on closed-source devices, LogicFuzz leverages vendor-exposed serial debugging to integrate structural (invocation-pattern) mutation with coverage-guided parameter mutation. It further unifies network behavior, internal system status, and physical actuation feedback into a multi-source anomaly-detection oracle capable of identifying failures beyond conventional crash-based signals.

We evaluate LogicFuzz on six commercial PLC models from three vendors. It successfully generated valid seeds for 88.47% of the targeted instructions and uncovered 19 instruction-level bugs, including four previously undisclosed vulnerabilities. These flaws span both device-control and system-management instructions, with several capable of forcing PLC shutdowns—posing direct physical-safety risks. By exploiting enable signals for precise control and reset, and by combining SDG-based structural mutation with coverage-guided parameter mutation, LogicFuzz delivers an instruction-centric fuzzing workflow for real-world PLC platforms.

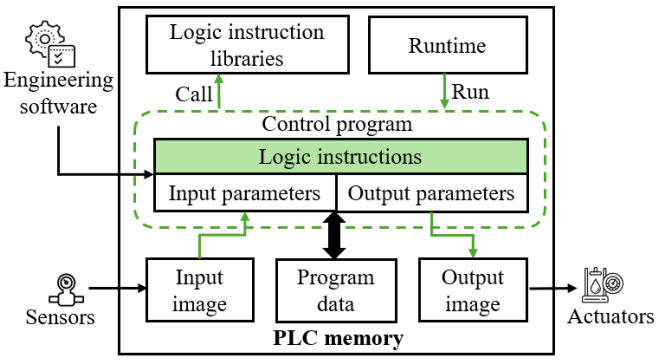

Fig. 1: Execution workflow of PLC logic instructions.

The remainder of this paper is organized as follows. Section II provides background on PLC logic instructions and motivating observations. Section III presents an overview of the LogicFuzz framework. Sections IV-VI detail the static analysis pipeline, seed program generation, and instruction-level fuzzing. Section VII reports our evaluation results. Section IX discusses insights and limitations. Section VIII surveys related work, and finally, Section X concludes the paper.

II. Background

This section describes how logic instructions are executed in PLCs, identifies two primary classes of instruction vulnerabilities, and explains how the enable-signal mechanism motivates our design.

A. Runtime Execution of Logic Instructions

Under the IEC 61131-3 standard, control logic can be written in five languages—Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL), and Sequential Function Chart (SFC). In this work, we focus on ST and its Siemens variant, SCL. A typical program executes in three phases. Parameter initialization declares variables and data types. Context initialization prepares run-time state by invoking auxiliary instructions or assigning values. Finally, instruction invocation executes the target logic instruction. This structure conforms to vendor usage constraints. By injecting externally supplied protocol data into the instruction's parameters, each fuzzing iteration remains aligned with the PLC's scan cycle, enabling us to exercise every instruction within its native execution context.

When powered on, the PLC loads firmware that manages peripherals, internal resources, and communication interfaces, and then enters a continuous scan cycle (illustrated by the green dashed box and arrow in Figure 1). At the beginning of each cycle, sensor readings are sampled into the *input image*. The control program then fetches operands—either from the input image or from the program-data region—dispatches the corresponding logic instructions from the instruction library, and executes them. The resulting values are written to the

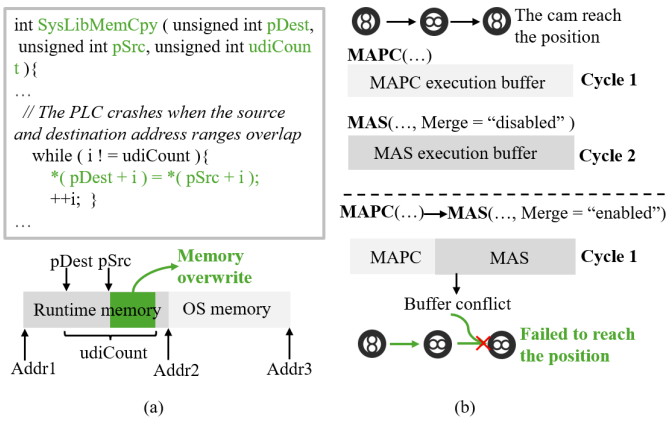

Fig. 2: Logic-instruction bugs are primarily triggered by (a) parameter values and (b) invocation patterns.

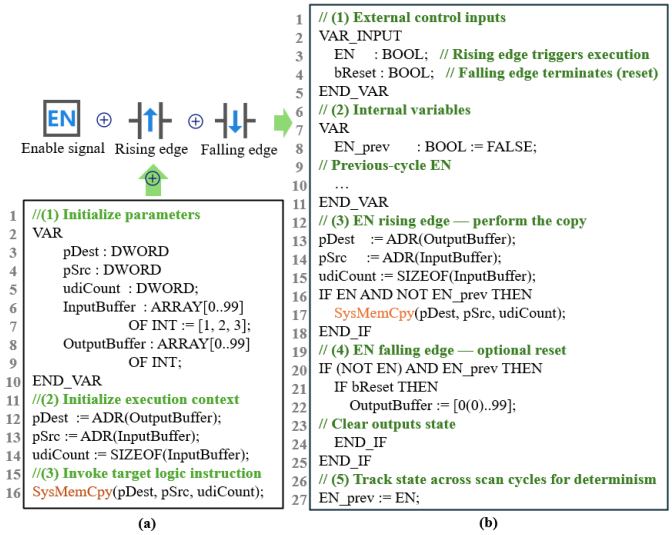

Fig. 3: Two control-program structures for `SysMemCpy`: (a) a representative production-style program; (b) our enable-signal-driven test program.

output image or back to program data; the output image is subsequently mapped to actuators, producing physical effects. During runtime, engineers can feed mutated inputs through the parameter read-write protocol and observe anomalies in multiple channels, including runtime logs, status LEDs, UART output, and Joint Test Action Group (JTAG) traces. Collectively, these signals expose crashes, scan cycle stalls, register corruption, and other instruction-level faults, providing a comprehensive oracle for fuzzing.

B. Vulnerable Logic Instructions

Logic-instruction bugs are firmware-level defects in the implementation of PLC logic instructions. When exercised under vendor-specified contexts and parameter constraints, these flaws can lead to memory corruption, denial-of-service, or unstable physical outputs. Such bugs typically fall into four categories: 1) input-handling errors (e.g., missing type or bounds checks), 2) resource-management faults (e.g., unchecked memory operations), 3) security or access control lapses (e.g., incorrect privilege settings), and 4) business-logic or configuration mistakes (e.g., inadequate functional-safety rules). We observe that vulnerabilities across all four categories are usually triggered by two factors: Parameter values—edge cases or adversarial data, and invocation patterns—the call order and inter-parameter dependencies that govern run-time interactions. Figure 2 (a) illustrates a value-driven flaw: `SysMemCpy` invokes `SysLibMemCpy` without validating either its source (`pSrc`) or destination (`pDest`) pointers. If `pDest` < `pSrc`, the routine overwrites memory and compromises the PLC.

Figure 2 (b) shows a pattern-driven flaw in Rockwell's ControlLogix 5570 PLC: Enabling the *merge* option allows the stop command `MAS` to pre-empt an unfinished cam-control command `MAPC`, corrupting its buffer and crashing the controller so that the axis halts short of its intended target. These cases suggest that exposing latent instruction-level bugs requires systematic mutation of both parameters and invocation relationships. LogicFuzz adopts this dual strategy by interleaving SDG-based semantic mutation with coverage-guided parameter mutation, enabling vulnerability discovery across diverse execution contexts.

C. Motivation

We refer to the control programs used during fuzzing as test programs. Early experiments with naive layouts revealed that many logic instructions are highly sensitive to PLC runtime resources—including memory buffers, file descriptors, network sockets, timers, persistent variables, and task objects. Consider fuzzing `SysMemCpy`, a memory-resource-sensitive instruction. As shown in Figure 3 (a), if the routine fails to release memory resources, the stale state carries over into subsequent fuzzing rounds. The consequences include false positives, masked bugs that require a clean start to manifest, distorted coverage metrics, and non-reproducible crashes. In contrast, session-oriented instructions (e.g., HTTP services) depend on persistent state and therefore must not be reset each cycle. A reset strategy must therefore be selective and aligned with the PLC scan cycle semantics. Resources are safely cleared only in the following cycle—not the current one—to avoid forcing slower PLCs into premature resource release.

Most vendors embed an enable signal in every logic instruction. The PLC samples this Boolean flag once per scan, and engineers can toggle it over the network to skip or re-trigger the instruction without modifying the program. Leveraging this mechanism, we make each test program both controllable and resettable—even for PLCs lacking a native enable signal—by introducing two external inputs, `EN` and `bReset` (Figure 3.b), together with rising- and falling-edge logic: **Rising edge** (`EN` transitions from `FALSE` to `TRUE`)—fires the instruction exactly once per test case execution. All rising-edge regions share the same `EN`. **Falling edge** (`EN` transitions from `TRUE`

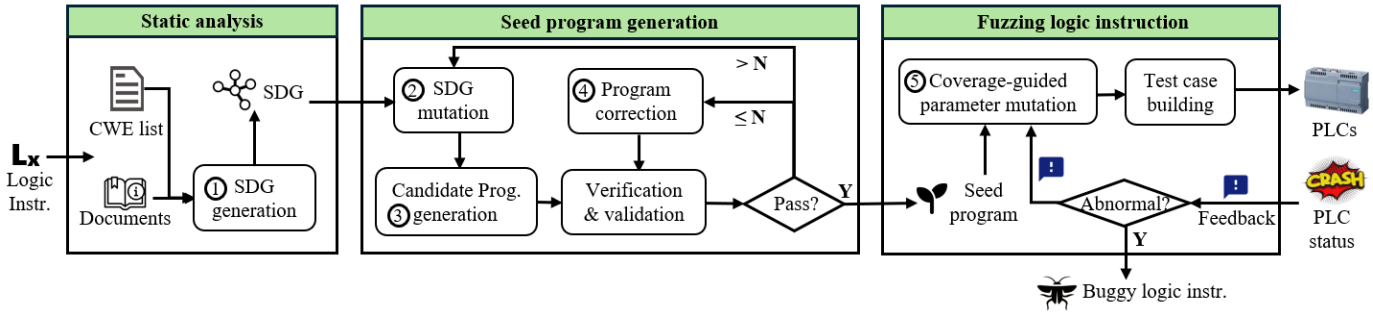

Fig. 4: Overview of LogicFuzz's workflow.

Bug clue extraction prompt

[Input] : Manuals, CWE list, L_x
[Task]:
For each CWE entry, assess whether its root cause and triggering conditions are applicable to logic instruction L_x using the provided manuals.
If applicable, generate one or more concise bug clues.
[Output]:
Each item in bug clues must be a dictionary with the following fields:

- "bug_type": a short CWE-relevant bug category.
- "trigger_param": the name of a parameter of *Lx* that is likely to trigger the bug.
- "trigger_condition": a suspicious value range or usage condition that may lead to the bug.

Extracted bug clues

```
"bug_clues":
[{ "bug_type": "unvalidated pointer dereference",
   "trigger_param": "pSockAddr",
   "trigger_condition" :"points to an invalid or NULL address"},
...]
```

Fig. 5: LLM prompt for bug-clue extraction.

to `FALSE`)—checks `bReset`; if `TRUE`, it reinitialises all internal state (buffers, counters, handles), ensuring a pristine environment for the next cycle. All falling-edge regions share the same `bReset`. Finally, the current `EN` value is latched for the subsequent scan to enable precise, cycle-synchronous control throughout fuzzing.

III. Overview

Figure 4 illustrates LogicFuzz's three-stage workflow. In the first stage, a static preprocessing pass merges vendor documentation with a CWE corpus to construct a semantic dependency graph (SDG) that captures each instruction's call ordering—including both caller-callee relations and execution sequencing—as well as its parameter dependencies. For each target instruction, LogicFuzz samples an SDG subgraph, mutates it by shuffling calls or rewiring dependencies, and supplies the mutated subgraph—together with a program-generation prompt—to a large language model, which synthesizes a candidate test program. Each generated program is then deployed to the tested PLC for verification. If the program passes validation, it becomes a seed; if it fails, LogicFuzz repeats the mutate–generate–verify loop until success or until a user-defined limit N is reached, after which the system starts with a fresh SDG mutation.

During the fuzzing stage, coverage-guided parameter mutation—informed by both the SDG and prior execution feedback—generates new inputs for each seed, which are executed directly on the physical PLC. A monitoring component continuously collects runtime logs, status LED states, and serial output. This feedback simultaneously guides subsequent mutations and serves as an anomaly-detection oracle for crashes, watchdog resets, and silent state corruption.

IV. Static Analysis

This section explains how LogicFuzz uses static analysis to build a Semantic Dependency Graph (SDG) that captures instruction-usage semantics. The SDG encodes (i) the call order—which covers caller–callee relations and execution sequencing—and parameter dependencies between the target instruction L_x and other instructions, (ii) per-parameter usage constraints, and (iii) bug clues distilled from the CWE corpus and vendor materials (manuals and example code).

LogicFuzz iterates over Common Weakness Enumeration (CWE) entries collected from MITRE CWE [26]. For each entry—represented by an example, root cause, and potential impact—it queries a large language model (LLM) with the CWE description and the manuals of L_x (prompt shown in Figure 5) to determine whether the CWE is applicable. We segment manuals by instruction index and initially feed only the section corresponding to the target instruction; cross-instruction segments are then added incrementally as needed to detect inter-instruction dependencies while respecting token limits. If the LLM deems a CWE applicable, it returns a bug clue that specifies the candidate bug type and, for each parameter of L_x, the conditions or values likely to trigger it. In addition, LogicFuzz utilizes the LLM to derive per-parameter usage constraints from the manuals, formalized as `(parameter, type, description)`.

Using an existing Structured Text (ST) AST extractor [39], LogicFuzz parses the example code of L_x, splits the statement sequence at control-transfer points (`IF`/`ELSE`, loop headers, `RETURN`/ `EXIT`) to form basic blocks, and assembles a control-flow graph (CFG). It then performs a standard iterative reaching-definitions analysis on the CFG to recover def–use

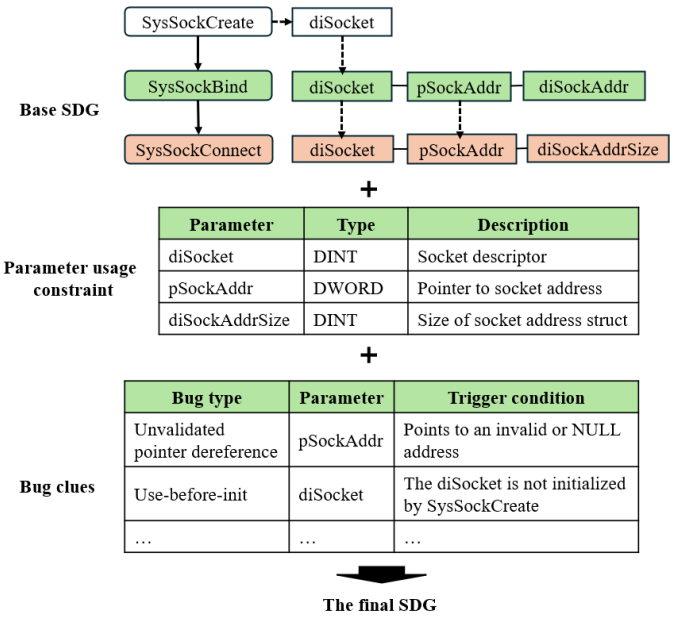

Fig. 6: Final SDG for `SysSockConnect`. In the base SDG, dashed edges denote data dependencies, while solid edges represent call and execution flow. Rounded rectangles indicate logic instructions, and rectangular nodes indicate instruction parameters.

relationships. The analysis follows the classical forward data-flow formulation of reaching definitions: For each basic block B, we iterate

$$\mathrm{IN}_B = \bigcup_{P \in \mathrm{pred}(B)} \mathrm{OUT}_P \tag{1}$$

$$\mathrm{OUT}_B = \mathrm{GEN}_B \cup (\mathrm{IN}_B \setminus \mathrm{KILL}_B) \tag{2}$$

to a fixed point, yielding the set of definitions that reach the entry of every block.

During a linear scan of the CFG, each AST node of the form `AssignStmt(... CallExpr(callee="`L_k`", args=[a_1, a_2]))` induces the following SDG edges: a) A *caller-callee edge* from the current caller to L_k. b) For every variable argument $v \in \{a_i\}$, a set of *parameter-dependency* edges $d \xrightarrow{(v)} L_k$ for all reaching definitions $d \in \mathrm{IN}_B(v)$; literal arguments are instead recorded as edge attributes without creating data-dependency edges. and c) An *execution-order* edge $L_{\text{prev}} \to L_k$, where L_{prev} is the most recent instruction call encountered in program order within the same basic block. The collected caller-callee, parameter-dependency, and execution-order edges form the basic SDG. Finally, LogicFuzz attaches the extracted bug clues and the per-parameter usage constraints to the corresponding nodes and edges to obtain the full SDG. An example SDG for `SysSockConnect` is shown in Figure 6. The SDG comprises three components: the basic SDG structure, parameter-usage constraints, and bug clues ("+" denotes conjunction). The basic SDG specifies that `SysSockConnect` requires prior invocations of

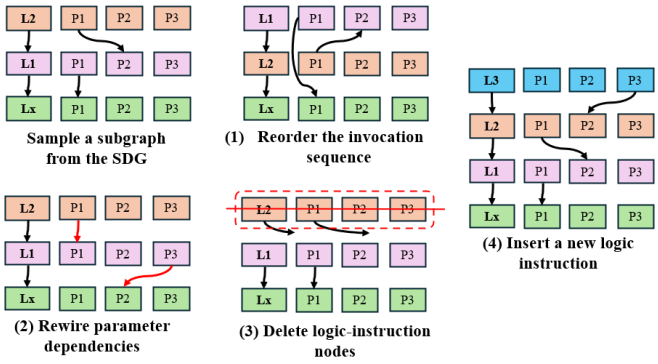

Fig. 7: Four SDG mutation operators. L_i denotes a logic instruction, and P_i denotes an instruction parameter.

`SysSockCreate` and `SysSockBind` to create and bind a socket before establishing a connection. The parameter-usage constraints capture the types and usage semantics of the instruction's three parameters, while the attached bug clues indicate the LLM-identified parameter values or conditions that may trigger potential vulnerabilities.

V. Seed Program Generation

Seed program generation proceeds in three steps: LogicFuzz first mutates the semantic dependency graph (SDG), then synthesizes a candidate program from the mutated subgraph, and finally executes a verify–correct–validate loop to obtain a compilable, semantics-conformant seed.

A. SDG Mutation

LogicFuzz begins by sampling a subgraph g from the target logic instruction L_x's SDG. If a synthesized program fails verification or validation, LogicFuzz randomly applies one of four mutation operators (Figure 7) to expose L_x to diverse execution contexts:

- **Reorder**: Permutes the invocation path leading to L_x while preserving each instruction's associated parameter-dependency edges.
- **Rewire**: Preserves call order but redirects parameter-dependency edges to different instructions and/or argument positions.
- **Delete**: Removes selected instruction nodes and their incident edges. Predecessors and successors retain their original ordering, while affected parameter edges become temporarily unbound.
- **Insert**: Samples a new instruction from the global pool, inserts it at a random position in g without altering existing dependencies, and randomly attaches its parameter edges to surrounding nodes.

B. LLM-based Candidate Program Synthesis

Given the mutated subgraph g and the full SDG, LogicFuzz queries an LLM (prompt in Figure 8) to synthesize a candidate test program $\mathcal{T}$ following the enable-signal–based template:

$$\mathcal{T} = \langle t_{\text{in}}, t_{\text{var}}, t_{\uparrow}, t_{\downarrow}, t_{\text{st}} \rangle . \tag{3}$$

Candidate program generation prompt

[Input] : g', L_x , SDG

[Instruction]
Please generate an IEC 61131-3 Structured Text test program $\mathcal{T}$ for the target logic instruction L_x based on the provided SDG. The program must strictly follow the program structure and requirements below:

Program structure: $\mathcal{T} \triangleq \langle A_{in}, A_{var}, A_{\uparrow}, A_{\downarrow}, A_{st} \rangle$

- $A_{in} \triangleq$ (EN : BOOL , bReset : BOOL , $\mathfrak{P}$ = { $p_1 \ldots p_n$ })
- $A_{var} \triangleq$ (EN_prev : BOOL := FALSE , Ctx : Σ)
- $A_{\uparrow}$ (rising edge) := {guard : EN $\wedge$ $\neg$EN_prev ; action: $\Phi(g', \mathfrak{P})$), where $\Phi(g', \mathfrak{P}) \triangleq$ Seq $\{L_i(Args_i) \mid (v_i, Args_i) \in \text{topo}(g')\}$
- $A_{\downarrow}$ (falling edge) :={guard : $\neg$EN $\wedge$ EN_prev; action: if bReset then Reset(Ctx) fi}
- A_{st} (state tracking) := { action: EN_prev $\leftarrow$ EN}

Fig. 8: LLM prompt for candidate-program synthesis.

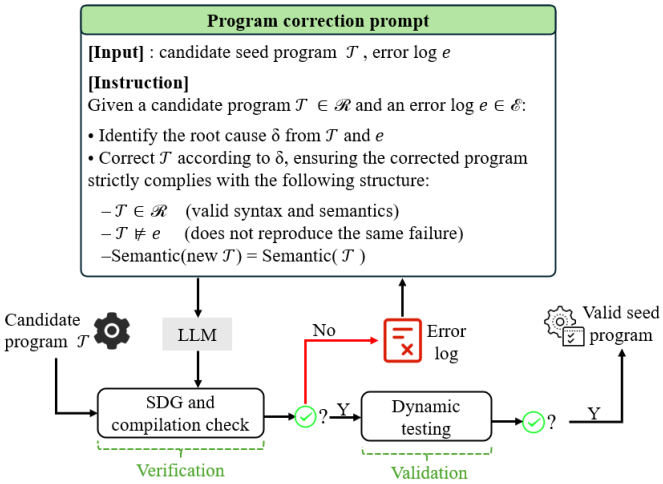

Fig. 9: Program verification-correction-and-validation workflow.

Inputs $t_{\text{in}} \triangleq (\texttt{EN} : \text{BOOL}, \texttt{bReset} : \text{BOOL}, \mathfrak{P} = \{p_1, \ldots, p_n\})$ declares the enable signal, reset flag, and all external parameters required by instructions in g, ensuring parameter completeness and successful compilation. *State* $t_{\text{var}} \triangleq (\texttt{EN\_prev} : \text{BOOL} := \text{FALSE}, \texttt{Ctx} : \Sigma)$ maintains cross-cycle context, where `EN_prev` supports edge detection and `Ctx` stores subgraph-specific state (e.g., buffers, handles, counters). *Rising-edge block* $t_{\uparrow} \triangleq \{\text{guard} : \texttt{EN} \wedge \neg\texttt{EN\_prev};\ \text{action} : \Phi(g, \mathfrak{P})\}$ fires exactly once per cycle and executes $\Phi(g, \mathfrak{P}) \triangleq \text{Seq}\{L_i(\text{Args}_i) \mid (v_i, \text{Args}_i) \in \text{topo}(g)\}$, where $\text{topo}(g)$ is a topological order of g and Args_i are resolved from $\mathfrak{P}$. *Falling-edge block* $t_{\downarrow} \triangleq \{\text{guard} : \neg\texttt{EN} \wedge \texttt{EN\_prev};\ \text{action} : \text{if } \texttt{bReset} \text{ then } \texttt{Reset(Ctx)} \text{ fi}\}$ conditionally reinitializes internal state to support both persistent-state and immediate-cleanup fuzzing modes. A final *State latching* block t_{st} records the current `EN` value into `EN_prev`, ensuring strict alignment with PLC scan-cycle semantics.

C. Program Verification and Correction

The LLM-generated program $\mathcal{T}$ may contain syntactic or semantic defects and must be validated before use. LogicFuzz combines compilation, SDG conformance checking, and automated correction into a unified verification loop (Figure 9). LogicFuzz first compiles $\mathcal{T}$. If compilation fails, the compiler error log e is sent to the correction prompt to guide the LLLM in revising $\mathcal{T}$. The revision must satisfy:

1) **Syntactic and semantic correctness:** $\mathcal{T} \in \mathcal{R}$, where $\mathcal{R}$ is the IEC 61131-3 ST grammar and semantic constraint set [38].
2) **Elimination of the reported failure:** $\mathcal{T} \not\models e$.
3) **Preservation of program semantics:** $\text{Semantic}(\text{new } \mathcal{T}) = \text{Semantic}(\mathcal{T})$.

LogicFuzz retries compilation and correction up to a limit of $N = 5$. If compilation still fails, $\mathcal{T}$ is discarded. LogicFuzz performs the reaching-definitions analysis from Section IV to extract the basic SDG g' from $\mathcal{T}$. It then checks equivalence between g' and the mutated subgraph g—including node sets, call ordering, and parameter-dependency relations. Non-conforming programs are discarded.

Program validation. LogicFuzz next performs short-term dynamic testing to confirm $\mathcal{T}$'s functional correctness. The program is deployed to the PLC, a live connection is established, and predefined test inputs are executed:

$$\mathbf{I}_i = \langle \texttt{EN}_i,\ \texttt{RESET}_i,\ \texttt{PARAM}_i \rangle.$$

For each input, LogicFuzz defines an invalidation oracle:

$$\text{oracle}_i = F_{\text{log}_i} \cup F_{\text{param}_i} \cup F_{\text{coverage}_i} \cup F_{\text{snapshot}_i},$$

where: F_{log_i} is the runtime log produced by engineering tools, F_{param_i} records observed input/state values, F_{coverage_i} is derived from PC traces via the serial port or JTAG (Section VI-A), F_{snapshot_i} captures PLC state snapshots before and after execution (e.g., memory, handles, timers).

If any predicate in oracle_i is triggered, $\mathcal{T}$ is deemed functionally invalid and discarded. Only programs that pass all oracles are accepted as seeds $\mathcal{T}_s$.

We employ four inputs and corresponding oracles:

- $\mathbf{I}_0 = \langle \text{True}, \text{False}, \emptyset \rangle$ (baseline rising-edge skeleton). Invalidation: log contains "error/crash/failure"; unexpected parameter/state changes; empty coverage; unchanged snapshots.
- $\mathbf{I}_1$ Normal invocation with valid parameters. Invalidation: identical to $\mathbf{I}_0$ plus: no expected state updates; no coverage increase; unchanged snapshot (indicating L_x did not execute).
- $\mathbf{I}_2$ Repeats $\mathbf{I}_1$ for one minute to expose latent leaks/timeouts; invalidation predicates identical to $\mathbf{I}_1$.
- $\mathbf{I}_3 = \langle \text{False}, \text{True}, \emptyset \rangle$ (falling-edge reset). Invalidation: reset effects absent; empty coverage; unchanged snapshot.

Only candidates that pass all runtime oracles are promoted as seeds for subsequent fuzzing.

VI. Logic Instruction Fuzzing

For each seed program $\mathcal{T}_s$, LogicFuzz performs parameter fuzzing to further explore bugs in the target instruction L_x.

Algorithm 1 Coverage-Guided Parameter Mutation

Require: Parameter set $\mathcal{P}$, mutation pool $\mathcal{M}$, subset size l, UCB constant C, log weight β

$cov \leftarrow 0$; $K \leftarrow 0$ ▷ Step 0: Initialize the global round counter.
for all $p_i \in \mathcal{P}$ **do** ▷ Step 0: Initialize the bandit stats.
  $n_i \leftarrow 0$; $R_i \leftarrow 0$
end for
while not STOP(K, cov) **do**
  for all $p_i \in \mathcal{P}$ **do** ▷ Step 1: Compute the UCB score.
    $\text{score}_i \leftarrow \text{UCBSCORE}(n_i, R_i, C, K)$
  end for
  $S_K \leftarrow \text{TOP}(\{\text{score}_i\}, l)$ ▷ Step 2: Select top-l parameters.
  for all $p \in S_K$ **do** ▷ Step 3: Parameter mutation.
    $m \leftarrow \text{RANDOMPICK}(\mathcal{M})$
    MUTATE(p, m)
  end for
  $(newCov, \log) \leftarrow$ EXECUTEANDGETCOVERAGE ▷ Step 4: Execute the test case and collect runtime feedback.
  $\Delta_{\text{cov}} \leftarrow newCov - cov$; $cov \leftarrow newCov$
  $\text{logScore} \leftarrow (\log = \varnothing)\,?\,0 : \text{SCORELLM}(\log)$
  $r \leftarrow \Delta_{\text{cov}} + \beta \cdot \text{logScore}$ ▷ Step 5: Compute blended reward
  for all $p_i \in S_K$ **do**
    $n_i \leftarrow n_i + 1$; $R_i \leftarrow R_i + \frac{r}{|S_K|}$
  end for
  $K \leftarrow K + 1$ ▷ Step 6: Update K.
end while

This section describes the design of our coverage-guided parameter mutation strategy, the enable- and reset-based execution control mechanism, and the runtime monitoring infrastructure.

A. Coverage-guided parameter mutation

Algorithm 1 adaptively steers mutations toward unexplored firmware regions with high bug potential by leveraging feedback from coverage traces (F_{coverage}) and runtime logs (F_{log}). At initialization, LogicFuzz maintains *per-parameter bandit statistics*—the selection count n_i and cumulative reward R_i—under a multi-armed bandit (MAB) model, where each parameter is treated as an arm. Mutation scheduling follows the Upper Confidence Bound (UCB) policy [3]:

$$\text{Score}_i = \frac{R_i}{n_i} + C \cdot \sqrt{\frac{2 \ln K}{n_i}}, \tag{4}$$

where the empirical mean $\frac{R_i}{n_i}$ captures a parameter's historical contribution to coverage gains or log anomalies, the second term encourages exploration, K denotes the current round, and C controls the exploration–exploitation trade-off. In each round, LogicFuzz selects the top-l parameters according to their UCB score.

For each selected parameter, LogicFuzz samples a mutation strategy from a pool comprising two behaviors: (i) *Random mutation*, which draws values from the parameter's full, type-specific domain (e.g., `INT` $\in [-32{,}768, 32{,}767]$), or strings as random byte sequences); and (ii) *bug-oriented mutation*, which uses the prompt in Figure 10 to instruct the LLM to alalyze L_x's SDG, including parameters, dependencies, and CWE-derived bug clues. The LLM interprets the bug clue and proposes values likely to trigger the vulnerability for the top-l parameters, falling back to safe default values if no triggerable inputs are identified.

Bug-oriented mutation prompt

[Input]: SDG, top-l parameters

[Task]:

1. Read the bug clues from the SDG
2. For each parameter, generate one or more test values that are likely to trigger the suspected bug based on the corresponding clue.
3. If generation fails, fall back to a safe default value.

Mutated top-l parameters of SysSockConnect

"CWE-119": {
"diAddressFamily": 2, "diType": 0,
"diProtocol": 10, "pSockAddr":
"127.0.0.1", "diSockAddrSize": 5000}

Fig. 10: LLM prompt for bug-oriented parameter mutation.

The generated values are applied via the online write protocol to construct a test case, which is then executed on the PLC. During execution, LogicFuzz collects F_{log} and F_{coverage} through the engineering software and via serial or Joint Test Action Group (JTAG) interfaces. Since fine-grained basic-block coverage is impractical for PLC firmware without heavy instrumentation [6], [18], we approximate coverage using *memory-block coverage*. PLC memory is partitioned into fixed-size blocks of 50 bytes, corresponding to the average basic-block size in PLC instruction libraries. At the end of L_x's execution cycle, LogicFuzz retrieves the program counter (PC) trace—a sequence of instruction addresses—via the serial port or JTAG interface. Each address is mapped to its corresponding memory block, and a block is marked as hit if at least one address falls within its range. Coverage is computed as:

$$\text{newCov} = \frac{1}{N_{\text{blk}}} \sum_{i=1}^{N_{\text{blk}}} \mathbb{1}(h_i > 0), \tag{5}$$

where N_{blk} is the total number of memory blocks, h_i denotes the hit count of block i, and $\mathbb{1}(h_i > 0)$ is the indicator function:

$$\mathbb{1}(h_i > 0) = \begin{cases} 1, & \text{if } h_i > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

To exploit semantic signals, LogicFuzz prompts the LLM (Figure 11) to analyze F_{log} and produce a likelihood score $\texttt{logScore} \in [0, 1]$ accompanied by a brief rationale. The prompt guides the LLM to: (i) analyze runtime logs collected from a real PLC, including resource usage, return codes, fault indicators, and abnormal patterns; (ii) assess whether the logs suggest internal failures or unexpected behavior; and (iii) output a confidence score with supporting textual reasoning.

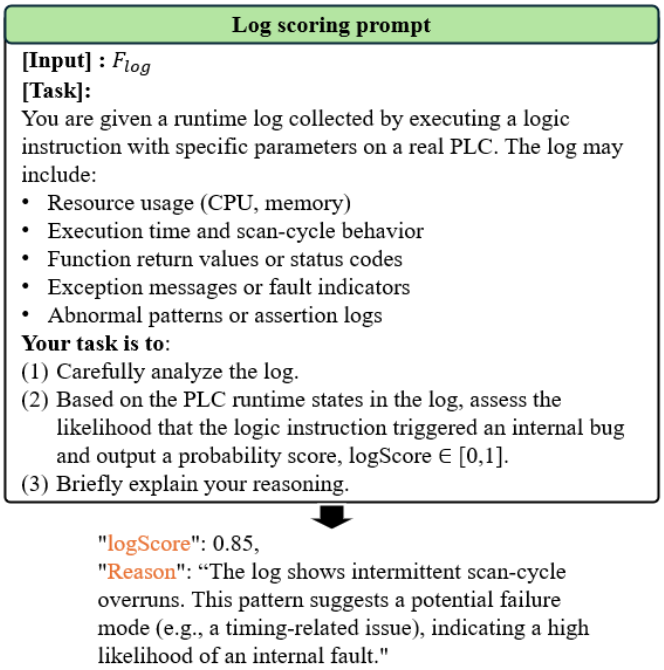

Fig. 11: LLM prompt for log scoring (*logScore* $\in [0, 1]$ estimates the likelihood that the logic instruction triggered an internal bug).

The round reward integrates both structural and semantic feedback:

$$r = \Delta_{\text{cov}} + \beta \cdot \texttt{logScore}. \tag{7}$$

To mitigate stagnation caused by deterministic logs and prevent logScore from dominating the reward, LogicFuzz randomizes $\beta \sim \mathcal{U}(0, 1]$, improving robustness across rounds. The reward r is evenly assigned to all mutated parameters to update their corresponding n_i and R_i. As K increases, the scheduler progressively prioritizes parameters that consistently improve coverage or elicit anomalous behavior, enabling deeper exploration of PLC execution paths.

B. Enable and Reset Control Mechanism

The enable signal `EN` and reset signal `bReset` are not subjected to mutation; instead, they serve as deterministic global controls governing the start, execution, and reset of each fuzzing test case. Before fuzzing begins, all control variables are initialized to `EN = FALSE` and `bReset = FALSE`. For each mutated test case, LogicFuzz first records a complete snapshot of the PLC state, including memory contents, system handles, and active timers. The fuzzer then sets signal `EN` to `TRUE`, triggering a single instruction execution.

After execution, LogicFuzz captures a second state snapshot and resets `EN` to `FALSE`. Based on a hierarchical reset policy, the fuzzer decides whether to assert `bReset = TRUE` to restore the PLC to a known state. A reset is triggered under any of the following conditions: (i) the monitoring module detects an anomaly, causing an immediate PLC restart; (ii) the pre- and post-execution snapshots are identical, indicating no observable state change; or (iii) the condition `resetScore = a·x + b·y > 0.5` holds, where `a` and `b` are LLM-inferred likelihoods that the instruction is resource-sensitive or session-persistent, respectively, and `x` and `y` are random values sampled from $[0, 1]$.

Upon receiving a reset signal, LogicFuzz invokes the reset module embedded in the seed program to restore the PLC state. If this reset fails—i.e., the pre- and post-reset snapshots remain unchanged—LogicFuzz escalates to a program-level reset using the engineering software's APIs to explicitly restore memory, handles, and timers. For example, LogicFuzz may invoke PLC memory-reset interfaces such as `SysMemSet` in CODESYS or `POKE_BLK` in Siemens environments to overwrite arbitrary memory regions. If the program-level reset also fails, the PLC is forcibly rebooted and fuzzing resumes from a clean state. This dual-layer reset design mitigates potential implementation errors in the reset module and ensures that the PLC reliably returns to a consistent, known state between fuzzing iterations.

TABLE I: Hardware and software configuration for evaluating LogicFuzz.

Category	Vendor	Detail
PLCs	Rockwell	CompactLogix 1756-L61 (firmware versions 16.023, 17.004, 19.015, 20.014)
		CompactLogix 1756-L33ER (firmware versions 20.011, 20.015, 24.011, 24.013)
	Siemens	S7-1200 (firmware versions 3.0.2, 4.3.2, 4.4.2)
		S7-1500 (firmware versions 1.5, 1.7, 2.9, 3.1)
	Wago	PFC 750-8203 (firmware versions 1.02.05, 02.03.09)
		758-870 (firmware version 3.00)
Engineering software	Rockwell	RSLogix 5000
	Siemens	TIA Portal V14
	Wago	CODESYS 2.3
Logic instructions	Rockwell	Total: 112 (External physical control: 77; Internal system operations: 34; Communication: 1).
	Siemens	Total: 126 (External physical control: 42; Internal system operations: 64; Communication: 20).
	Wago	Total: 100 (External physical control: 37; Internal system operations: 36; Communication: 27).
LLMs	OpenAI	GPT-4o
	Deepseek	Deepseek-R1
	Anthropic	Claude Sonnet 4

C. Anomaly Detection

LogicFuzz defines a set of monitoring oracles M_{oracle} derived from three complementary feedback channels—program runtime logs (F_{log}), PLC system indicators (F_{light}), and communication interfaces (F_{comm})—to detect crashes, hangs, and timeouts:

$$M_{oracle} = F_{\text{log}} \cup F_{\text{light}} \cup F_{\text{comm}}. \tag{8}$$

Guided by vendor documentation on execution failures, system faults, communication errors, and control malfunctions, we implemented 56 reusable oracles spanning three service categories: (1) External physical control (21 oracles), capturing hardware-module and CPU-level faults; (2) Commu-

TABLE II: Logic-instruction bugs discovered by LogicFuzz.

Bug ID	Logic Instr.	Function	Bug type	Bug triggering conditions	Affected PLC	Version	Security risk
Lgx169520	GSV	System operation	Lack of boundary checks	Set the WCT to year `1900`.	Rockwell 1756-L61	17.004, 19.015	Denial of Service
Lgx179778				Set the WCT to year `9999`.	Rockwell 1756-L61	20.014	Denial of Service
Lgx169520	SSV	System operation	Lack of boundary checks	Set the WCT to year `1900`.	Rockwell 1756-L61	17.004, 19.015	Denial of Service
Lgx179778				Set the WCT to year `9999`.	Rockwell 1756-L61	20.014	Denial of Service
IN25781	ALMA	System operation	Incorrect data type handling	Set `ALMA`'s tag length to `100`.	Rockwell 1756-L61	19.015	Memory corruption
Lgx135333				Set `ALMA`'s tag length to `-1`.	Rockwell 1769-L33ER	17.08	MNRF
IN25781	ALMD	System operation	Incorrect data type handling	Set `ALMA`'s tag length to `100`.	Rockwell 1756-L61	19.015	Memory corruption
Lgx135333				Set `ALMA`'s tag length to `-1`.	Rockwell 1769-L33ER	17.08	MNRF
Lgx00136317	MAJ	Physical control	Improper Parameter Initialization	Set `MergeSpeed` and `LockPosition` to `-1`.	Rockwell 1756-L61	20.013, 20.014	Denial of Service
New	MRP	Physical control	Unoptimized Logic	Set `CurrentPosition` + `MRPPosition` to exceed the maximum representable range.	Rockwell 1756-L61	16.023, 17.004	Denial of Service
CVE-2020-15782	MOVE_BLK _VARIANT	System operation	Lack of boundary checks	Make the `SRC` and `DEST` arrays overlap.	Siemens S7-1200	3.0.2, 3.3.4	Information exposure
					Siemens S7-1500	1.5	Information exposure
New	MOVE_BLK	System operation	Lack of boundary checks	Use overlapping `SRC` and `DEST` addresses.	Siemens S7-1200	3.0.2, 3.3.4	Information exposure
					Siemens S7-1500	1.5	Information exposure
WAGO-2021-01	SysMemCpy	System operation	Lack of boundary checks	Use overlapping `pSrc` and `pDest` addresses.	Wago 750-8203	3.0	Memory overflow
					Wago 758-870	1.02	Memory overflow
WAGO-2021-02	MemCpy	System operation	Lack of boundary checks	Use overlapping `pbSrc` and `pbyDest` addresses.	Wago 750-8203	3.0	Memory overflow
					Wago 758-870	1.02	Memory overflow
WAGO-2021-03	SysMemMove	System operation	Lack of boundary checks	Use overlapping `pSrc` and `pDest` addresses.	Wago 750-8203	3.0	Memory overflow
					Wago 758-870	1.02	Memory overflow
WAGO-2021-04	MemMove	System operation	Lack of boundary checks	Use overlapping `pSrc` and `pDest` addresses.	Wago 750-8203	3.0	Memory overflow
					Wago 758-870	1.02	Memory overflow
WAGO-2021-05	SysMemSet	System operation	Memory access violation	Out-of-bounds length (`udiCount`).	Wago 750-8203	3.0	Information exposure
					Wago 758-870	1.02	Information exposure
New	SysFileWrite	System operation	Unauthorized access	Out-of-bounds pointer (`buffer`).	Wago 750-8203	3.0	Code injection
					Wago 758-870	1.02	Code injection
New	SysFileRead	System operation	Unauthorized access	Out-of-bounds pointer (`buffer`).	Wago 750-8203	3.0	Information exposure
					Wago 758-870	1.02	Information exposure

nication services (19 oracles) covering 11 industrial protocols, including disconnects, timeouts, checksum mismatches, and station loss; and (3) Internal system services (16 oracles) monitoring file systems, operating system components, and internal databases. As an illustrative example, a Modbus anomaly is reported if *any* of the following conditions holds: (i) the module fails to respond or times out after a standard request (e.g., function code `0x03`); (ii) the device status LED indicates a network error (commonly red, depending on the PLC model); or (iii) the runtime log contains a "Modbus service error/exception."

For confirmed bugs, LogicFuzz performs root-cause analysis by replaying the proof-of-concept (PoC) input, recording PC traces via the serial interface, reconstructing the executed basic-block sequence, and reverse-engineering the corresponding firmware regions. This process enables manual auditing to verify that the vulnerability is consistently triggerable under the PoC's constraints.

VII. Evaluation

We first describe our experimental setup, then evaluate LogicFuzz's effectiveness on logic-instruction bugs, ablate five components, compare key design choices against prior work, and finally quantify the impact of exploitation. LogicFuzz consists of roughly 3,000 lines of Python and AutoIt.

A. Experiment Setting

We use `pdfplumber` [30] to convert PLC manuals [35] [4] [1] into machine-readable text for LLM consumption. All LLM-driven steps—bug-clue extraction for SDG construction, seed synthesis, and program correction—are implemented via the Python OpenAI library [28]. We import synthesized seed programs into vendor engineering suites (RSLogix5000, TIA Portal V14, and CODESYS 2.3) and use AutoIt to automate validation: `Send("^v")` pastes the program and `ControlClick` triggers verification. Our fuzzing engine is built on Boofuzz [5]. Following RLPatch [44] and Cojocar et al. [14], we locate JTAG pinouts on Rockwell and Siemens PLCs and use a Segger J-Link to extract debug traces. For WAGO PLCs, we enable root-level SSH access and use `perf` to collect PC traces during fuzzing. For WAGO, we serialize test cases using Boofuzz `Fields`. For Siemens and Rockwell, we deliver test cases via vendor-specific interfaces: `db_write` from Snap7 and `LogixDriver.write` from pycomm3, respectively.

As summarized in Table I, our evaluation covers 338 logic instructions from vendor manuals—112 Siemens, 126 Rockwell, and 100 WAGO—spanning external control, internal system services, and communications. We evaluate two PLC models per vendor across different firmware versions: WAGO PFC 750-8203 and 758-870; Siemens S7-1200 and S7-1500; and Rockwell CompactLogix 1756-L61 and 1756-L33ER. To ensure vendor-faithful compilation, deployment, and validation during fuzzing, we use each vendor's default engineering software: RSLogix5000 for Rockwell, TIA Portal V14 for Siemens, and CODESYS 2.3 (Wago Automation Alliance) for WAGO.

Unless otherwise noted, all static analysis, seed generation, and fuzzing agents use `gpt-4o-2024-11-20` with temper-

ature 0 for determinism. To respect the query budget, we cap the seed-correction loop at five iterations. For coverage-guided parameter mutation, we set the UCB exploration coefficient to $C = \sqrt{2}$ following Auer [3]. Each logic instruction is fuzzed for 12 CPU-hours.

B. Real-world Bugs Discovered by LogicFuzz

LogicFuzz tested 338 logic instructions, generated 3,294 seed programs, and raised 514 alerts during fuzzing. Across static analysis, seed generation, and fuzzing, LLM queries cost $146.31. Among the alerts, 147 were false positives: benign network-service responses (Modbug, TCP, DNP3, OPC) arrived after our preset alarm timeout and were misclassified as anomalies. We also observed 83 missed alarms because the anomaly detector failed to recognize the "Minor Fault" state string reported by RSLogix5000. After de-duplication and manual triage of the remaining 284 alerts, we confirmed 19 logic-instruction bugs across six PLCs, including four previously undisclosed vulnerabilities. We now analyze these bugs in detail.

As summarized in Table II, six bugs stem from missing boundary checks. For Rockwell, when LogicFuzz supplied wall-clock time (WCT) values corresponding to years 1900 and 9999 to `GSV` and `SSV`, RSLogix5000 reported a `Minor Fault`. Further analysis indicates that these instructions do not enforce appropriate upper/lower bounds on WCT inputs, causing faults on out-of-range yet syntactically valid parameters. An attacker can exploit such compliant-but-out-of-bounds parameters via the CIP protocol to trigger a denial-of-service (DoS) condition on the PLC. Likewise, `SysMemMove`, `MemMove`, `SysMemCpy`, and `Memcpy` accept overlapping or otherwise invalid pointers. Since the CODESYS manual does not clearly standardize the safety constraints of these instructions, LogicFuzz generated inputs that cause source and destination pointers to overlap, resulting in string overflows on two WAGO PLCs and leaking program/parameter addresses. On Siemens S7-1200 and S7-1500, `MOVE_BLK_VARIANT` and `MOVE_BLK` invoke `memcpy` without validating source/destination positions; the `ERROR` LED turned red before clearing. Without bounds checks, these cases can corrupt PLC memory—potentially halting actuation or causing DoS if control logic is affected—and may access memory regions outside the PLC program's execution sandbox, enabling information leakage.

Four defects in `ALMD` and `ALMA` arise from incorrect type handling. Both instructions default to treating label parameters as strings and interpret inputs `-1` and `100` as string lengths, leading to out-of-bounds accesses at addresses `-1` and `100`, respectively. The controller subsequently enters a Major Non-Recoverable Fault (MNRF)—halting operations, disabling outputs, and requiring manual recovery (e.g., firmware re-download) [44]. An attacker can leverage engineering-tool APIs to inject malicious payloads into the `ALMA`/`ALMD` parameters, making recovery costly and disrupting production.

We further identify a parameter-initialization flaw in `MAJ`. Although `MergeSpeed` and `LockPosition` are intended to be constrained to {`0`, `1`}, setting them to `-1` is not rejected by the engineering software; the PLC subsequently fails to process the value, leaving the controller suspended while the tool reports an `unknown fault`. Similarly, `MRP` fails to guard against integer overflow in its additive logic; overflowed results also suspend the PLC until restart, with RSLogix5000 reporting an `unknown fault`. Both issues enable a straightforward DoS vector.

Finally, `SysMemSet`, `SysFileWrite`, and `SysFileRead` do not validate target buffer pointers. For example, `SysLibMemSet` calls `memset` directly, and the file I/O routines only check `if (file)` without verifying buffer addresses. When LogicFuzz provided out-of-range pointers, WAGO PLCs crashed, disconnected from the engineering software, and showed a red `RUN` indicator. Such improper address/permissions handling risks unauthorized access to sensitive files or memory-resident secrets (e.g., PLC keys). Moreover, `SysFileWrite` allows writes to arbitrary paths outside the control program's directory. Based on the operator manual indicating that WAGO PLCs run Linux, LogicFuzz generated test cases that attempt to write probe files and malicious payloads into root-protected directories; our monitor detected content changes in protected paths and classified this behavior as an unauthorized-access vulnerability. An attacker could exploit this primitive to persist malicious artifacts on the PLC and potentially escalate to full device compromise.

C. Ablation Study

The ablation study quantifies the contribution of LogicFuzz's five components—SDG generation (①), SDG mutation (②), candidate-program generation (③), program correction (④), and coverage-guided parameter mutation (⑤)—via five configurations. Cfg 1 removes SDG generation (①); downstream stages receive only the instruction name and a correct invocation relation. Cfg 2 removes SDG mutation (②) but retains the initial SDG. Cfg 3 removes our structured candidate-generation procedure (③) and replaces it with a single generic prompt ("Generate a test program for the instruction using an enable signal."). Cfg 4 removes program correction (④). Cfg 5 removes coverage-guided mutation (⑤) and instead uses type-aware random parameters.

We evaluate each configuration using three metrics: (i) valid-seed ratio over the 338 instructions (counting both the seed program and its generated test cases), (ii) total alerts raised during fuzzing, and (iii) average coverage across the 338 programs. For each instruction, all six configurations (full + five ablations) terminate once the per-instruction LLM query budget of $0.05 is exhausted.

As shown in Table III, removing SDG generation (Cfg 1) sharply reduces the valid-seed ratio to 61.63% (vs. 88.47% for the full system) and more than halves the number of alerts (63 vs. 142). This indicates that SDG-derived context (e.g., bug clues and parameter-usage constraints) is critical for synthesizing semantically valid, fuzzable seeds. Disabling SDG mutation (Cfg 2) further collapses alerts to 44—far

TABLE III: Ablation results for LOGICFUZZ's five components.

Configuration	Valid Seed Ratio (%)	Alarms	Avg Coverage (%)
Cfg 1 (②,③,④,⑤)	61.63	63	16.43
Cfg 2 (①,③,④,⑤)	87.24	44	13.11
Cfg 3 (①,②,④,⑤)	31.52	8	16.83
Cfg 4 (①,②,③,⑤)	74.21	77	17.71
Cfg 5 (①,②,③,④)	86.17	36	9.37
LogicFuzz	88.47	142	22.33

below the full system's 142—suggesting that SDG mutation is a primary driver of "depth", i.e., reaching behaviors that trigger non-trivial failures. Notably, the valid-seed ratio also drops (87.24% vs. 88.47%), consistent with the generator being confined to the initial SDG and less able to escape semantic/syntactic dead ends; iterating SDG mutation with validation allows LogicFuzz to explore alternative but semantically consistent instruction relations and recover valid seeds.

Replacing structured candidate generation (Cfg 3) yields the lowest valid-seed ratio (31.52%), underscoring the importance of the fixed, enable-signal–aligned template in producing compilable, executable programs. Removing program correction (Cfg 4) decreases the valid-seed ratio to 74.21% ($\approx$ 14% absolute below full). In the full system, 47 seeds required one correction round, 15 required two, and 9 required three; none required four or five. Thus, the five-iteration cap is conservative, and correction also improves cost efficiency: the average LLM cost per successful seed is \$0.038 (LogicFuzz) versus \$0.079 (Cfg 4). Finally, removing coverage-guided mutation (Cfg 5) keeps the valid-seed ratio relatively high (86.17%) but reduces alerts to 36 and drops average coverage to 9.37% (vs. 22.33% full), indicating that unguided parameter exploration substantially limits test depth.

Overall, SDG generation and SDG mutation are the main drivers of reach and bug-finding depth; the structured template and correction primarily improve seed validity and cost efficiency; and coverage-guided mutation is critical for broad, deep exploration as reflected by coverage and alert yield.

D. Comparison with Existing Work

To the best of our knowledge, LogicFuzz is the first framework purpose-built for fuzzing PLC logic instructions. Prior work overlaps with individual modules (e.g., seed synthesis or feedback collection) but does not provide an end-to-end pipeline for instruction-level fuzzing. Where applicable, we adapt and extend these techniques into component-wise baselines to enable fair comparisons and to isolate the impact of LogicFuzz's design choices.

Static analysis. We compare LogicFuzz's static analysis against the knowledge-retrieval agent in Agent4PLC [24], an LLM-based control program generator. LogicFuzz extracts invocation relationships via AST traversal, whereas Agent4PLC relies on LLM-only semantic extraction. For fairness, we augmented Agent4PLC's retrieval prompt with: "Please extract the parameter dependencies, the calling relations, and execution orders for the given logic instruction," use its default retrieval context window (four 1K-character chunks), and provide both systems with the same manual. We then manually audit all 338 instructions. LogicFuzz recovers invocation relations for all 338 (100%), whereas Agent4PLC succeeds on 121. The gap is largely due to context truncation in Agent4PLC's limited session window, which causes the LLM to omit required calls, hallucinate missing relations, and misorder execution steps.

Across the 1,743 bug clues attached to LogicFuzz's SDGs, 98.71% fall into four categories: buffer overflows, out-of-bounds reads, integer overflows, and insufficient index validation. We attribute this distribution to two vendor-side tendencies: (i) emphasis on physical control logic that underspecifies finite bit-width constraints (e.g., overflow bounds and sign handling), and (ii) extensive use of raw pointers in internal services and communications without strict bounds enforcement, increasing the risk of out-of-bounds access and index/offset misuse.

Seed program generation. We evaluate seed generation capability by comparing LogicFuzz with Agent4PLC and PromptFuzz [25], adding the enable-signal mechanism to both baselines for fairness. For Agent4PLC, we rewrite the query as: "Generate a test program using the enable signal mechanism for the given logic instruction." For PromptFuzz, we substitute logic-instruction manuals for APIs, replace "API" with "logic instruction" in its prompts, and append enable-signal logic to produce executable programs. To ensure consistent correctness criteria across vendors and methods, we evaluate all generated seeds using LogicFuzz's unified correction, verification, and validation pipeline.

Each method generates one seed per instruction (338 total), and we measure the fraction that passes LogicFuzz's validation. To reduce backend-specific variance, we also test two code-oriented LLMs (DeepSeek [16] and Claude [2]). We additionally report average generation time and the average number of repair iterations to capture end-to-end generation cost. As shown in Table IV, LogicFuzz achieves a substantially higher pass rate than both baselines, highlighting the benefit of instruction-oriented semantic constraints. PromptFuzz produces $\approx$ 1% valid programs; its 14 successes correspond to elementary arithmetic/Boolean operations that require little semantic extraction. Agent4PLC's valid outputs similarly cluster in "light semantic, resource-independent" categories (arithmetic/logic, timers/counters, strings, type conversions), which do not require scan-cycle-aware state management and are well represented in pretraining corpora. For vendor-specific, resource-sensitive, and state-dependent instructions, even with enable-signal/ scan-cycle scaffolding, Agent4PLC often fails to reconstruct parameter dependencies and preconditions, and it frequently omits required pointer/handle initialization, reuse, or reset, yielding semantically invalid programs.

LogicFuzz also exhibits lower generation cost than Agent4PLC and PromptFuzz (9.10 seconds average generation time and 1.50 repair iterations), indicating that SDG-guided semantic constraints reduce invalid generations and unnecessary repair cycles. Table IV further shows clear backend

TABLE IV: Evaluation covers all 338 logic instructions. For each method, we report: (1) *Passed*/338 (pass rate), (2) *Time*/338 (average generation time), and (3) *Iters*/338 (average number of repair iterations). *Row Avg.* reports, for each model, the mean pass rate/time/iters across methods, while *Col. Avg.* aggregates each method across the three models.

Model	Agent4PLC			PromptFuzz			LogicFuzz			Row Avg.
	Pass (%)	Avg. Time (s)	Avg. Iters	Pass (%)	Avg. Time (s)	Avg. Iters	Pass (%)	Avg. Time (s)	Avg. Iters	
GPT-4o	28.00%	14.72	2.21	0.89%	17.83	4.62	**92.90%**	**7.32**	**1.44**	40.60% / 13.29/2.76
DeepSeek-R1	21.89%	20.38	2.37	2.07%	32.17	4.93	**89.94%**	**10.87**	**1.57**	37.97% / 21.14/2.96
Claude Sonnet 4	25.15%	17.06	2.19	1.18%	24.09	4.71	**91.98%**	**9.11**	**1.49**	39.44% / 16.75/2.80
Col. Avg.	25.01%	17.39	2.26	1.38%	24.70	4.75	**91.61%**	**9.10**	**1.50**	39.34% / 17.06/2.84

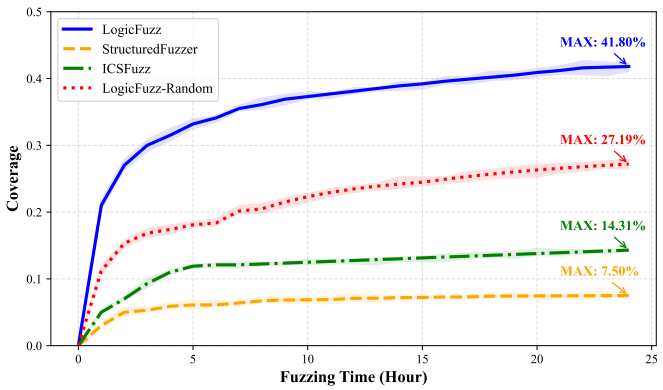

Fig. 12: Coverage comparison of LogicFuzz, StructuredFuzzer, and ICSFuzz over 24 hours. The solid line denotes mean coverage, and the shaded band indicates the min-max range.

differences: `GPT-4o` attains the highest average pass rate (40.60%), together with the lowest average time cost (13.29 seconds per program) and the fewest repair iterations (2.76 per program), motivating its use as our default backend.

Mutation strategy. To assess LogicFuzz's mutation strategy, we compare it with ICSFuzz [40] and StructuredFuzzer [22], two recent control-program fuzzers that mutate instruction/function block parameters. These baselines represent different philosophies: StructuredFuzzer uses uninformed random value generation, whereas ICSFuzz prioritizes parameters whose mutations are associated with new path discovery. To fairly adapt them to logic-instruction fuzzing, we retain LogicFuzz's static analysis, seed generation, and test-case construction, and replace only the coverage-guided parameter mutation policy with each baseline. Concretely, StructuredFuzzer becomes type-consistent random sampling; for ICSFuzz, we replace its "new path discovered" trigger with LogicFuzz's "new memory block hit" condition derived from memory-block coverage. We additionally disable bug-oriented parameter mutations and keep only random parameter mutations, forming LogicFuzz-Random to isolate the contribution of LLM/SDG-guided semantic mutations.

We run 24-hour campaigns for each of the 338 instructions and evaluate all methods using LogicFuzz's coverage metric. As shown in Figure 12, LogicFuzz reaches high coverage substantially faster than both baselines; within the first six hours, it exceeds 33.46%, while ICSFuzz and StructuredFuzzer grow slowly and plateau at 14.31% and 7.50%, respectively. Moreover, LogicFuzz continues to increase coverage throughout the full 24 hours without clear convergence, suggesting a stronger capability to penetrate deeper execution behavior over time. When disabling bug-oriented mutations (LogicFuzz-Random), coverage drops to 27.19%, indicating that LLM/SDG-guided semantic mutations materially improve mutation effectiveness and long-term coverage gain.

TABLE V: Average per-test execution time (**T**) and average memory usage (**M**) by PLC and tool.

PLC Model	LogicFuzz		LogicFuzz-GUI		LogicFuzz-ICS		LogicFuzz-Quartz	
	T (s)	M (%)	T (s)	M (%)	T (s)	M (%)	T (s)	M (%)
Wago 750-8203	0.111	6.81	3.321	32.54	0.016	10.17	0.00041	7.89
Wago 758-870	0.106	6.88	3.893	37.88	0.026	11.31	0.00076	8.13
Siemens S7-1200	0.125	8.19	8.613	43.14	N/A	N/A	N/A	N/A
Siemens S7-1500	0.118	7.69	9.121	47.23	N/A	N/A	N/A	N/A
Rockwell 1756-L33ER	0.110	12.25	6.337	52.21	N/A	N/A	N/A	N/A
Rockwell 1756-L61	0.086	13.44	7.813	49.73	N/A	N/A	N/A	N/A

Execution performance. We evaluate test case execution performance against three extensible baselines—ICS3fuzzer [19], ICSFUZZ [40], and ICSQuartz [41]. We implement three execution backends: (i) LogicFuzz-GUI, built on ICS3Fuzzer and using AutoIt to automate vendor engineering suites for parameter passing and execution; (ii) LogicFuzz-Quartz, running on a Linux host and orchestrating instruction-level fuzzing on WAGO PFC PLCs by hooking the CODESYS build-and-deploy pipeline; and (iii) LogicFuzz-ICS, extending ICSFuzz to execute parameter passing by intercepting the memory-mapping path on Linux-based WAGO PFC PLCs. We measure two metrics: average per-test execution time and memory usage. For each of the 338 instructions, we select one seed and execute one test case on all six PLCs, record the total time, and compute the mean (**T**) by dividing by 338; we also record the memory usage (**M**) after each case.

Table V shows that on WAGO 750-8203 and 750-870, LogicFuzz is slower than the memory-based LogicFuzz-ICS and host-based LogicFuzz-Quartz. However, LogicFuzz operates on any PLC that supports upload, download, and validation via vendor engineering software, whereas ICSFUZZ and ICSQuartz are constrained to WAGO-series devices due to platform-specific interfaces. In terms of memory, LogicFuzz

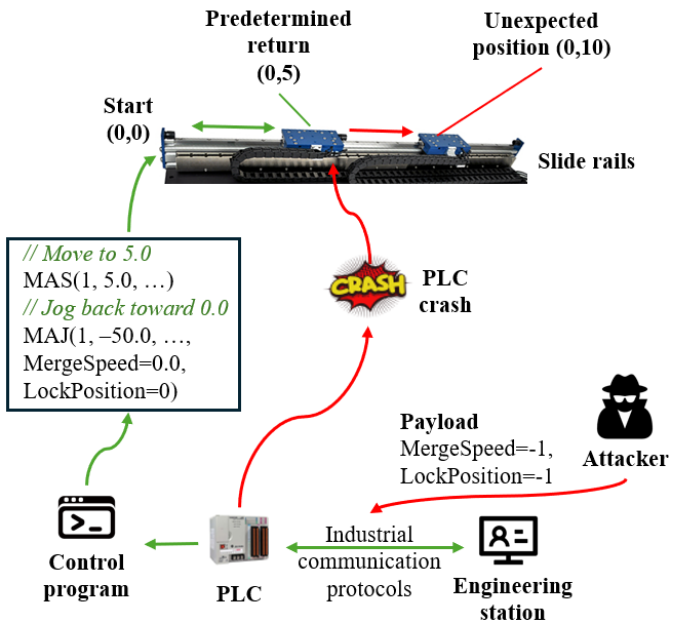

Fig. 13: Example exploitation of a Rockwell `MAJ` logic-instruction bug. The green arrows depict the normal slide-rail workflow, while the red arrows show the adversary's attack flow.

has the smallest footprint across all methods: its usage ranges from 6.81% to 13.44%, consistently lower than LogicFuzz-GUI, LogicFuzz-ICS, and LogicFuzz-Quartz on the same PLC models. Overall, LogicFuzz provides strong portability and scalability with favorable resource efficiency, making it well suited for long-running, large-scale, and parallel fuzzing campaigns.

E. Effect of Exploitation

This section demonstrates how the parameter-initialization flaw in `MAJ` can be weaponized and the physical impact it can induce. Figure 13 shows our experimental setup: a linear-rail platform representative of robotic-arm motion stages and conveyor transport. An engineer first downloads the control program to the PLC from an engineering workstation. The workstation then monitors and adjusts the rail state by mapping logic-instruction parameters to PLC memory over common fieldbus/industrial Ethernet protocols (e.g., Modbus RTU and EtherNet/IP). The program moves the rail from the home position $(0, 0)$ to the return point $(0, 5)$ using the `MAS` move command, and then jogs it back to $(0, 0)$ using a `MAJ` loop executed under default safety parameters (`MergeSpeed = 0.0`, `LockPosition = 0`), which enforces smooth and bounded motion.

In many deployments, industrial protocols are operated without encryption, strong authentication, or fine-grained write controls. Consequently, an adversary with network access can inject a single Modbus write that overwrites the PLC memory locations holding `MAJ`'s parameters, setting `MergeSpeed = -1` and `LockPosition = -1`. These illegal values trigger the firmware bug; in our experiment, the PLC crashes and the rail overshoots to $(0, 10)$, beyond its mechanical limit. Unlike classic control-logic injection [42]—which typically requires replacing the full program or spoofing multi-step protocol exchanges—this exploit flips only one instruction's parameters, generating minimal traffic and thus offering a substantially stealthier attack surface. Moreover, because it abuses a firmware-level defect rather than application logic, the same technique can apply across PLC models and firmware versions that share the vulnerable instruction implementation.

Protocol safeguards such as encryption, integrity protection, mutual authentication, anti-replay, and variable whitelisting can raise the bar against packet-tampering-based exploitation of logic-instruction vulnerabilities. However, an attacker may still satisfy these stronger preconditions—for example, by obtaining valid credentials/keys, compromising the engineering workstation, or gaining physical access to the PLC—thereby restoring the feasibility of parameter overwrite and increasing the likelihood of successful exploitation.

VIII. Related Work

In this section, we focus on fuzz testing for embedded firmware and PLC systems and the use of large language models (LLMs) in program synthesis and fuzz testing.

A. Firmware and PLC Fuzzing

Firmware-level bug discovery is typically dominated by three families of techniques—static taint analysis, symbolic execution, and fuzzing on simulated/emulated images—but each is ill-suited to systematically testing PLC *logic instructions*. Static taint analysis, exemplified by DTaint [12], reconstructs potentially dangerous data flows by combining data-structure similarity with bottom-up call-chain tracing. In PLC firmware, however, logic instructions heavily rely on runtime-generated data and vendor-specific execution contexts, which hinders accurate context recovery and leads to excessive false positives. Symbolic execution frameworks such as Angr [34] lift firmware to an intermediate representation and explore paths symbolically. In practice, this incurs severe path explosion and memory overhead, often requires substantial manual effort to retrace and validate findings, and still produces many false positives—limitations that are amplified by the complex, proprietary runtimes of commercial PLCs. Finally, simulation- or emulation-based firmware fuzzers such as Fuzzware [33] typically do not support the processor architectures or peripheral models used in most PLCs; even when binaries can be loaded, low simulation fidelity and frequent execution failures make these approaches ineffective for exercising logic-instruction implementations. Collectively, these constraints motivate a dedicated, instruction-aware approach such as LogicFuzz.

Existing PLC fuzzing research targets three layers—industrial communication protocols, control programs, and the PLC runtime—yet none systematically addresses instruction-level bugs (Table VI). Protocol fuzzers such as ICS3Fuzzer [19] and PCFuzzer [23] mutate packets

or protocol fields to uncover implementation flaws in proprietary stacks. Since their mutation logic is coupled to protocol grammars, they cannot capture the invocation order or parameter dependencies that govern logic-instruction behavior, and thus provide little leverage for instruction-centric testing. Runtime fuzzers, represented by Sizzler [20] and FieldFuzzer [8], manipulate live external inputs (e.g., network traffic, physical I/O, or direct memory writes) to influence a running PLC. However, they neither infer instruction usage patterns nor synthesize semantically consistent call sequences and parameters, leaving instruction-level faults largely unexplored. Control-program fuzzers such as ICSFuzz [40], ICSQuartz [41], and StructuredFuzzer [22] aim to expose higher-level logic faults by perturbing control flow and state transitions in user code. While some approaches (e.g., StructuredFuzzer) do mutate instruction parameters, they typically rely on rule-based or random heuristics that ignore fine-grained parameter dependencies and invocation constraints. As a result, generated tests frequently violate semantic preconditions and fail to reach the boundary conditions and deep behaviors required to stress logic-instruction implementations. In summary, prior work advances fuzzing at the protocol, runtime, and control-program layers, but lacks the specialized semantics, guided mutations, and hardware-aware execution model needed for systematic *instruction-level* fuzzing.

B. LLMs for Program Synthesis and Fuzzing

LLMs have evolved from statistical language models into general-purpose systems capable of instruction following and multi-domain reasoning. Models such as GPT-3 [7] and GPT-4 [27] leverage large-scale pretraining to produce fluent, coherent outputs over long contexts, while alignment techniques such as reinforcement learning from human feedback (RLHF) [29] improve their ability to follow user intent and generate reliable responses. This prompt-based interaction paradigm has enabled LLM applications beyond traditional NLP, including code synthesis and completion [10], scientific knowledge encoding [37], automated test generation, failure discovery, and analysis of industrial control logic [24]. These developments suggest that LLMs can serve as task-oriented agents that combine domain knowledge with general reasoning—an attractive capability for generating semantically valid test programs.

Fuzzing logic instructions requires repeatedly generating programs that respect instruction semantics and execute correctly under PLC scan-cycle constraints. As summarized in Table VI, existing PLC program generators are typically rule-based or model-based. Rule-based systems such as PLCspecif [15] and G4LTL-ST [11] encode fixed logic units, contracts, and behavioral patterns and map them to task specifications. However, instruction semantics vary widely across vendors and firmware versions, making comprehensive rule maintenance quickly infeasible. Recent LLM-driven approaches such as Agent4PLC [24] generate generic control programs via retrieval-augmented generation, but they are not designed for instruction-level fuzzing. In particular, they omit (i) fuzzing-relevant semantics such as invocation order, parameter constraints, and bug-trigger conditions; (ii) hardware-aware logic synchronized with scan cycles and I/O mappings; and (iii) mechanisms that deliberately drive boundary conditions. Consequently, the generated programs are often unsuitable as fuzzing seeds and fail to exercise instruction-level corner cases.

TABLE VI: Comparison of related work. *Sup.* indicates support for logic-instruction fuzzing: (●) direct, (⊙) partial, and (○) none.

Related work	Method	Representative works	Sup.
Control program generation	Rule-based	G4LTL-ST (2014) [11]	●
		PLCspecif (2016) [15]	●
	Model-based	Agents4PLC (2024) [24]	⊙
API fuzz-driver generation	Rule-based	APICraft (2021) [43]	○
	Model-based	PromptFuzz (2023) [25]	○
PLC fuzzing	PLC protocol fuzzing	ICS3FUZZER (2021) [19]	○
		PCFuzzer (2022) [23]	○
	PLC program fuzzing	ICSFuzz (2021) [40]	○
		StructuredFuzzer (2024) [22]	⊙
		ICSQuartz (2025) [41]	○
	PLC runtime fuzzing	Sizzler (2023) [20]	○
		FieldFuzzer (2023) [8]	○

API fuzz-driver generation resembles an instruction-fuzzing workflow and exists in both rule/model-based variants [43] [25]. However, these approaches generally overlook the syntax and execution model of PLC control programs. The resulting drivers often fail to compile or to be downloaded and executed in vendor toolchains, and their structure diverges from logic-instruction test programs: they lack parameter-to-I/O mappings and do not implement scan-cycle interactions with external inputs and outputs. Finally, generic API mutation strategies do not encode the semantic constraints of logic instructions, limiting their effectiveness for instruction-level fuzzing.

IX. Discussion

This section discusses LogicFuzz's limitations, results, and future directions from three perspectives: manual effort, robustness, and scalability.

A. Manual Effort

LogicFuzz requires modest human effort that is largely one-time. We collected seven vendor manuals, 473 logic-instruction code samples (from official sites, GitHub, and forums), and one CWE list in about 1 hour. We drafted a 2,000-word prompt in 2 minutes and implemented roughly 3K lines of code in 6 hours, including 600 lines for program validation and anomaly-detection oracles (about 2 hours). Setting up serial and Joint Test Action Group (JTAG) connectivity between six PLCs and their engineering suites took 1 hour. Using a traffic monitor, we extracted vendor-specific engineering software protocols to automate parameter passing (another 1 hour). Finally, manual triage of 514 alerts required 4 hours. Except for alert triage, these tasks—documentation

collection, prompt drafting, implementation, communication setup, and protocol extraction—are reusable investments (e.g., for future fuzzing of engineering-software protocols). Because of the soundness—completeness trade-off, some manual confirmation is unavoidable; nevertheless, LOGICFUZZ achieves 55.25% precision (284/514). Our next goal is to tighten the oracles to reduce false alarms and further lower review effort.

B. Robustness

LOGICFUZZ's automatic test-program generation depends on two factors. (i) *Industrial knowledge completeness.* While vendor documentation can omit details, such cases are uncommon: among the 473 collected instructions, only 12 (2.54%) lacked sufficient parameter-usage constraints to build a complete SDG. All 338 instructions evaluated in our study include code examples and usage descriptions detailed enough to support fully automatic SDG extraction. (ii) *LLM capability.* Combining SDG-derived constraints with a fixed-structure prompt yields an 88.47% seed-generation success rate. Alternative backends exhibit noticeable variance, suggesting that stronger code-capable LLMs can further improve correctness and reduce repair effort.

A related limitation is that test cases derived from CWE patterns inevitably reflect known bug classes. In contrast, LOGICFUZZ's SDG-mutation operators and type-aware random parameter generation are pattern-agnostic, enabling exploration beyond CWE templates and improving the chance of uncovering previously unknown instruction-level faults.

At the same time, LOGICFUZZ cannot fully control LLM nondeterminism. In our evaluation, 11.63% of seeds failed due to: (i) hallucinations that misinterpret logic instructions as other languages and generate non-IEC 61131-3 code (3.72%); (ii) unexpected characters (e.g., undecodable bytes) (2.37%); and (iii) failure to follow specified parameter ranges or instruction-call order (5.54%). Future work will refine prompts and explore fine-tuning to mitigate issue (i) and to further reduce reliance on SDG and other industrial knowledge. We will also introduce comprehensive static checks over program syntax and instruction parameters to detect and reject issues (ii) and (iii) before deploying test cases.

C. Scalability

As shown in Table V, LOGICFUZZ maintains low and stable overhead across six PLCs: average per-test execution time remains within 0.086–0.125 s and memory usage within 6.81%–13.44%. Despite heterogeneous engineering suites and runtime environments across WAGO, Siemens, and Rockwell, per-instruction cost remains stable, indicating that expanding to additional PLC families increases total cost approximately linearly rather than being dominated by any single platform.

Architecturally, LOGICFUZZ scales by decoupling instruction-level logic from device-specific adapters. SDG construction, semantic parsing, and parameter-constraint extraction are vendor-agnostic; adding a new PLC primarily requires implementing thin adapters for the engineering software and monitoring interface, while reusing the existing seed-generation and mutation pipeline. Similarly, when the instruction corpus grows beyond 338, LOGICFUZZ incrementally parses new manuals and adds SDG nodes without modifying the core fuzzing loop.

LOGICFUZZ currently relies on PLC debug interfaces that expose PC traces or equivalent runtime visibility (e.g., JTAG-based tracing or `perf`-based sampling), which are commonly available on commercial PLCs. For restricted devices without JTAG/SSH/engineer-mode access, viable alternatives include: (1) static firmware rewriting or instrumentation (where permitted) to insert lightweight coverage probes or trace hooks that map internal execution paths to observable events; and (2) using vendor engineering suites (e.g., TIA Portal or CODESYS) through their APIs to retrieve runtime states or snapshots for approximate coverage estimation and anomaly detection. In future work, we plan to unify heterogeneous coverage and state signals into a vendor-independent metric, reducing reliance on specific debug interfaces while preserving effective coverage guidance on more constrained PLC platforms.

X. CONCLUSION

This paper presents *LogicFuzz*, the first automated fuzzing framework designed to uncover logic-instruction bugs across heterogeneous PLC platforms. To generate test cases that respect instruction semantics, *LogicFuzz* constructs a Semantic Dependency Graph (SDG) that captures invocation relations, parameter constraints, and bug clues. We develop an AST-based extraction pipeline for SDG construction and successfully generate SDGs for all 338 evaluated instructions. Inspired by field engineers' programming practices, we design an enable-signal–based fuzzing harness that is controllable and resettable across scan cycles, and we craft structured prompts that guide an LLM to synthesize seed programs. In experiments, *LogicFuzz* achieves an 88.47% seed-generation success rate. To target the two primary triggers of logic-instruction bugs—invocation patterns and input parameters—we propose SDG-guided mutation operators that perturb instruction invocation relationships. We further introduce a memory-block–based coverage metric obtained via serial and Joint Test Action Group (JTAG) interfaces and couple it with UCB-guided parameter mutation. Together, these complementary mutation strategies drive deeper exploration and enable *LogicFuzz* to uncover 19 bugs across six PLCs, including four previously undisclosed vulnerabilities. Finally, we build anomaly-detection oracles that combine runtime logs, communication probes, and PLC indicator lights, achieving 55.25% monitoring precision. Looking ahead, we plan to refine generation strategies and explore model fine-tuning to further improve seed validity while reducing reliance on SDG-style industrial knowledge, thereby increasing the efficiency of logic-instruction bug discovery.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful and constructive feedback. This work was supported in part by the National Natural Science Foundation of China (NSFC)

under Grant No. 92467201, NSFC under Grant No. 62472302, NSFC under Grant No. 62402225, and by the project "Intelligent Identification and Risk Response System for Data Ransomware Attack on Industrial Internet Enterprises" under Grant No. 0747-2361SCCZA193.

ETHICS CONSIDERATIONS

We responsibly disclosed all discovered logic-instruction bugs to Siemens, WAGO, and Rockwell, who approved their publication in this paper. We note that the vendors had previously identified and remediated 15 logic-instruction bugs (assigned internal IDs) in firmware updates released before our disclosure; these fixes address the affected versions. After we reported four previously unknown logic-instruction bugs, WAGO, Siemens, and Rockwell indicated that they plan to address them in future PLC firmware releases.

REFERENCES

[1] 3S Smart Software Solutions. 3s smart software solutions becomes codesys gmbh. https://www.codesys.com/news-events/press-releases/article/3s-smart-software-solutions-gmbh-becomes-codesys-gmbh.html, 2021.

[2] Anthropic. Claude: An ai assistant by anthropic. https://www.anthropic.com/index/claude, 2023. Accessed: 2025-07-22.

[3] P. Auer and R. Ortner. Ucb revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, 61(1-2):55–65, 2010.

[4] Rockwell Automation. Automation system design software. https://www.rockwellautomation.com/en-us/products/software/factorytalk/designsuite/studio-5000.html, 2025.

[5] Boofuzz Project. Boofuzz: A network protocol fuzzing framework for humans. https://boofuzz.readthedocs.io/en/stable/index.html, 2025.

[6] M. Börsig, S. Nitzsche, M. Eisele, R. Gröll, J. Becker, and I. Baumgart. Fuzzing framework for esp32 microcontrollers. In *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6. IEEE, 2020.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[8] 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*, pages 499–512, October 2023.

[9] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. Electricity Information Sharing and Analysis Center (E-ISAC), 388.1–29, 2016. p. 3.

[10] M. Chen. Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374, 2021.

[11] C. H. Cheng, C. H. Huang, H. Ruess, and S. Stattelmann. G4ltl-st: Automatic generation of plc programs. In *International Conference on Computer Aided Verification*, pages 541–549, Cham, July 2014. Springer International Publishing.

[12] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, June 2018.

[13] Claroty Team82. The race to native code execution in plcs: Using rce to uncover siemens simatic s7-1200/1500 hardcoded cryptographic keys. https://claroty.com/team82/research, 2024.

[14] L. Cojocar, K. Razavi, and H. Bos. Off-the-shelf embedded devices as platforms for security research. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[15] D. Darvas, E. Blanco Viñuela, and I. Majzik. Plc code generation based on a formal specification language. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. IEEE, 2016.

[16] Deepseek. Deepseek. https://www.deepseek.com/, 2025.

[17] A. Di Pinto, Y. Dragoni, and A. Carcano. Triton: The first ics cyber attack on safety instrument systems. In *Proceedings of Black Hat USA*, pages 1–26, 2018.

[18] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 5(1):18, 2022.

[19] D. Fang, Z. Song, L. Guan, P. Liu, A. Peng, K. Cheng, and L. Sun. Ics3fuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 849–860, December 2021.

[20] 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*, 19:1660–1671, 2023.

[21] D. Formby, S. Durbha, and R. Beyah. Out of control: Ransomware for industrial control systems. In *RSA Conference*, volume 4, 2017.

[22] K. A. Koffi, V. Kampourakis, J. Song, C. Kolias, and R. C. Ivans. Structuredfuzzer: Fuzzing structured text-based control logic applications. *Electronics*, 13(13):2475, 2024.

[23] P. Liu, Y. Zheng, Z. Song, D. Fang, S. Lv, and L. Sun. Fuzzing proprietary protocols of programmable controllers to find vulnerabilities that affect physical control. *Journal of Systems Architecture*, 127:102483, 2022.

[24] Z. Liu, R. Zeng, D. Wang, G. Peng, J. Wang, Q. Liu, and W. Wang. Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents. *arXiv preprint arXiv:2410.14209*, 2024.

[25] Y. Lyu, Y. Xie, P. Chen, and H. Chen. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, pages 3793–3807, December 2024.

[26] MITRE. CWE-699: Software Development. https://cwe.mitre.org/data/definitions/699.html, 2025. CWE View, Version 4.18.

[27] OpenAI. Gpt-4 technical report. https://openai.com/research/gpt-4, 2023. Accessed: 2025-08-07.

[28] OpenAI. Openai python api library. https://pypi.org/project/openai/, 2025.

[29] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[30] pdfplumber. pdfplumber python library (0.11.5). https://pypi.org/project/pdfplumber/, 2025.

[31] Langner R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[32] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, and G. Vigna. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*, pages 484–500. IEEE, May 2021.

[33] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, and A. Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256. USENIX Association, 2022.

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: State of The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[35] Siemens. *Siemens S7-1200 PLC Programmable Manual*. Siemens, 2025. https://cache.industry.siemens.com/dl/files/465/36932465/att_106119/v1/s71200_system_manual_en-US_en-US.pdf.

[36] J. Slowik. Evolution of ics attacks and the prospects for future disruptive events. Threat Intelligence Centre, Dragos Inc., 2019.

[37] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, N. Hartshorn, E. Saravia, N. Goyal, D. Dohan, C. Dyer, R. Taori, et al. Galactica: A large language model for science. https://arxiv.org/abs/2211.09085, 2022.

[38] M. Tiegelkamp and K.-H. John. *IEC 61131-3: Programming Industrial Automation Systems*, volume 166. Springer, Berlin, Germany, 2010.

[39] TUMAIS. Iec611313antlrparser. https://github.com/TUMAIS/IEC611313ANTLRParser.

[40] D. Tychalas, H. Benkraouda, and M. Maniatakos. ICSFuzz: Manipulating i/o and repurposing binary code to enable instrumented fuzzing in ics control applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2847–2862, 2021.

[41] C. Villa, C. Doumanidis, H. Lamri, P. H. N. Rajput, and M. Maniatakos. Icsquartz: Scan cycle-aware and vendor-agnostic fuzzing for industrial control systems. In *Network and Distributed System Security (NDSS) Symposium*, 2025.

[42] H. Yoo and I. Ahmed. Control logic injection attacks on industrial control systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 33–48. Springer, 2019.

[43] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu. Apicraft: Fuzz driver generation for closed-source sdk libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828, 2021.

[44] M. Zhou, H. Wang, K. Li, H. Zhu, and L. Sun. Save the bruised striver: A reliable live patching framework for protecting real-world plcs. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1192–1207, 2024.