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

Jiaxing Cheng, Ming Zhou, Haining Wang, Xin Chen, Yuncheng Wang,
Yibo Qu, Limin Sun

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

南京理工大学
NANJING UNIVERSITY OF SCIENCE & TECHNOLOGY

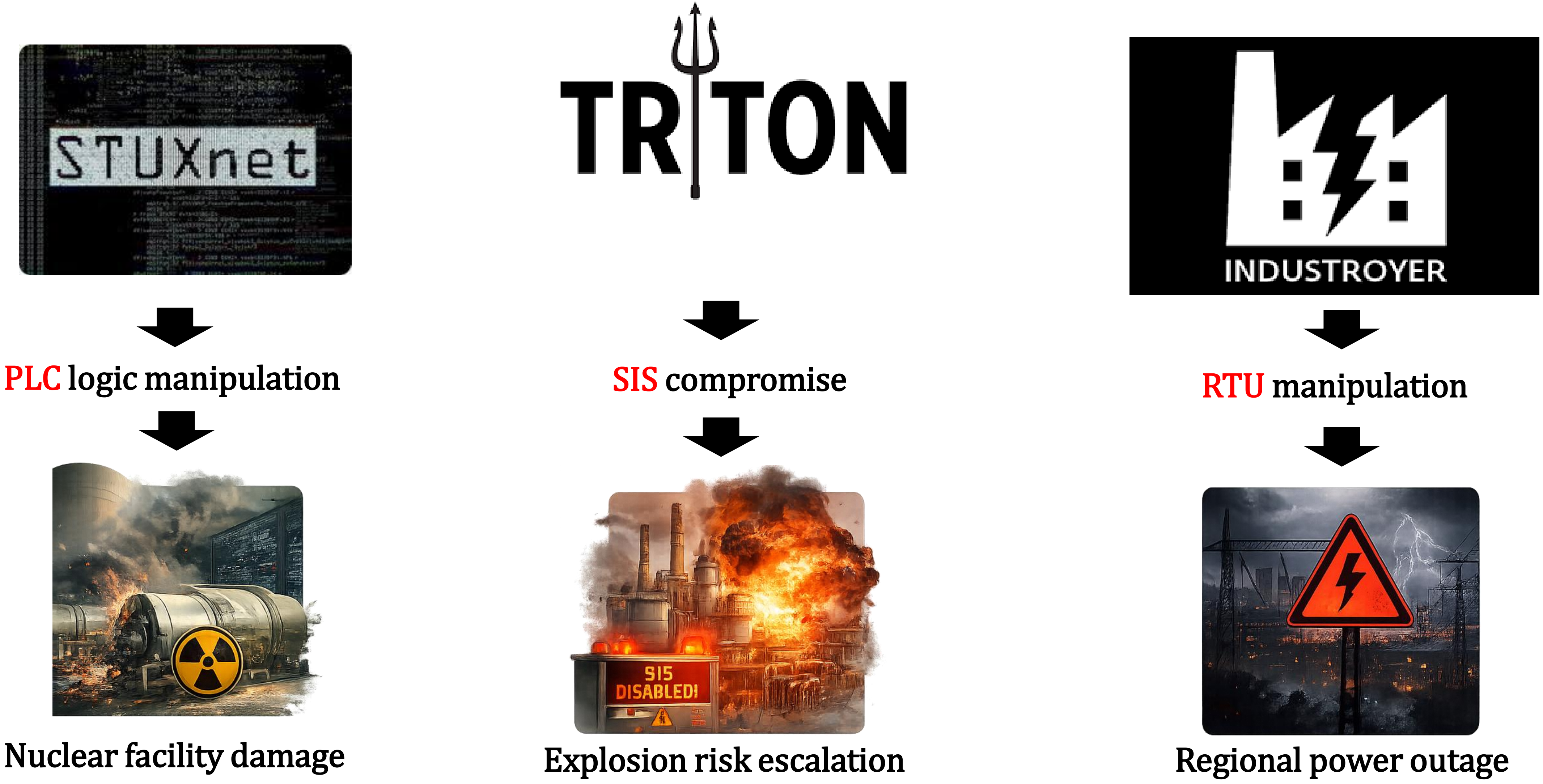
VT VIRGINIA TECH

中国科学院大学
University of Chinese Academy of Sciences

NDSS
SYMPOSIUM

# Background

The disruption or compromise of ICS may cause physical damage; but how do malware cross cyber space to physical impact? — Compromise the control devices



PLC logic manipulation

Nuclear facility damage

SIS compromise

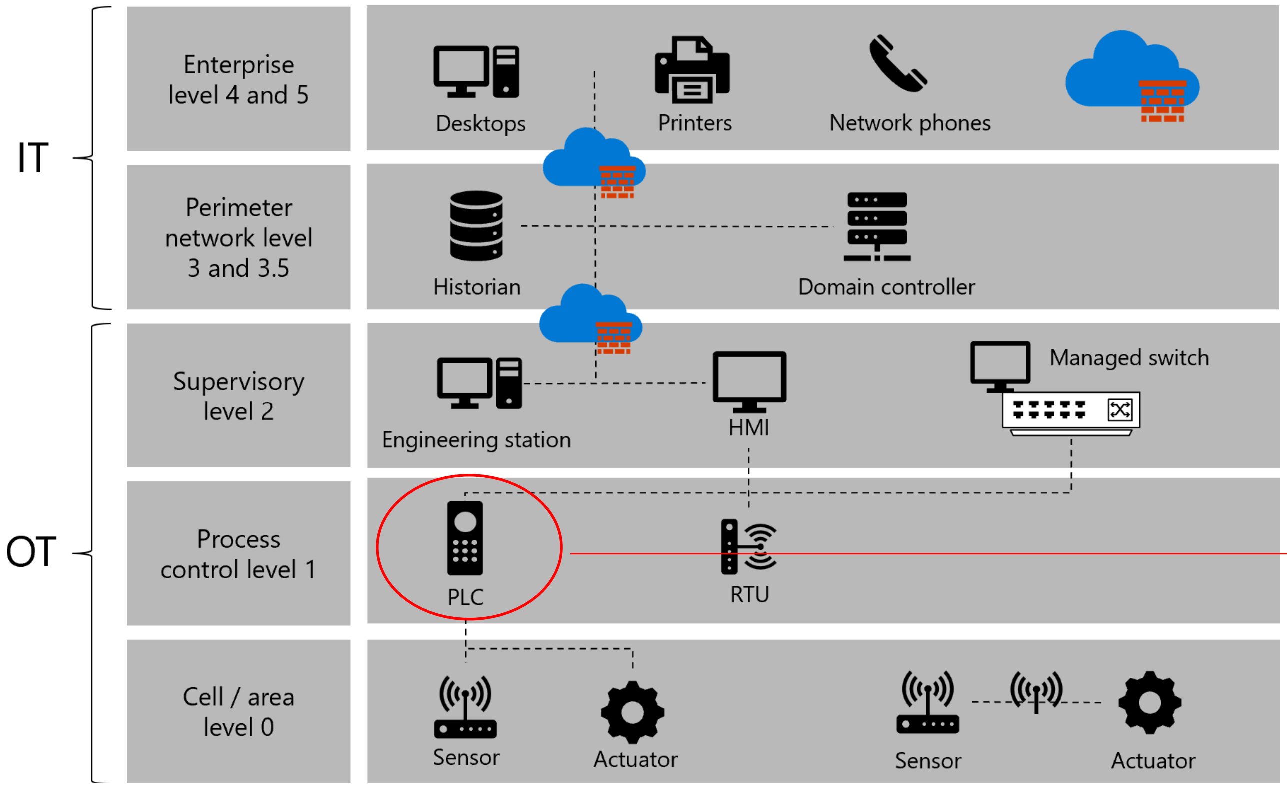
Explosion risk escalation

RTU manipulation

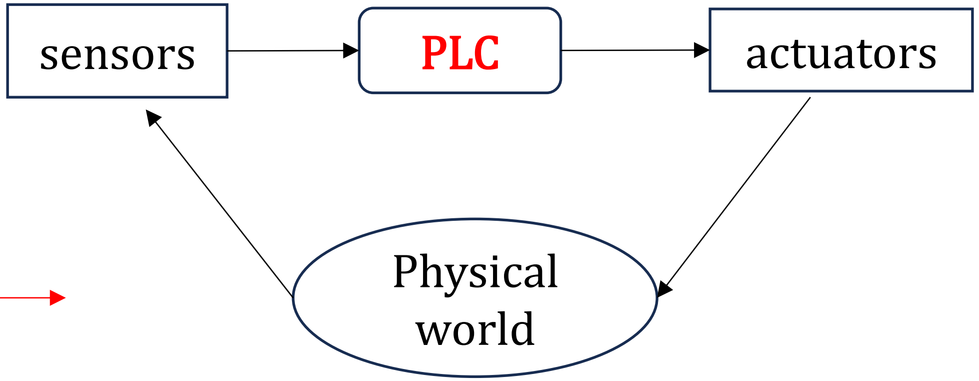
Regional power outage

# Background

## The core control devices of ICS — Programmable Logic Controller (PLC)



ICS architecture


sensors

PLC

actuators

Physical world

PLC workflow

# Background

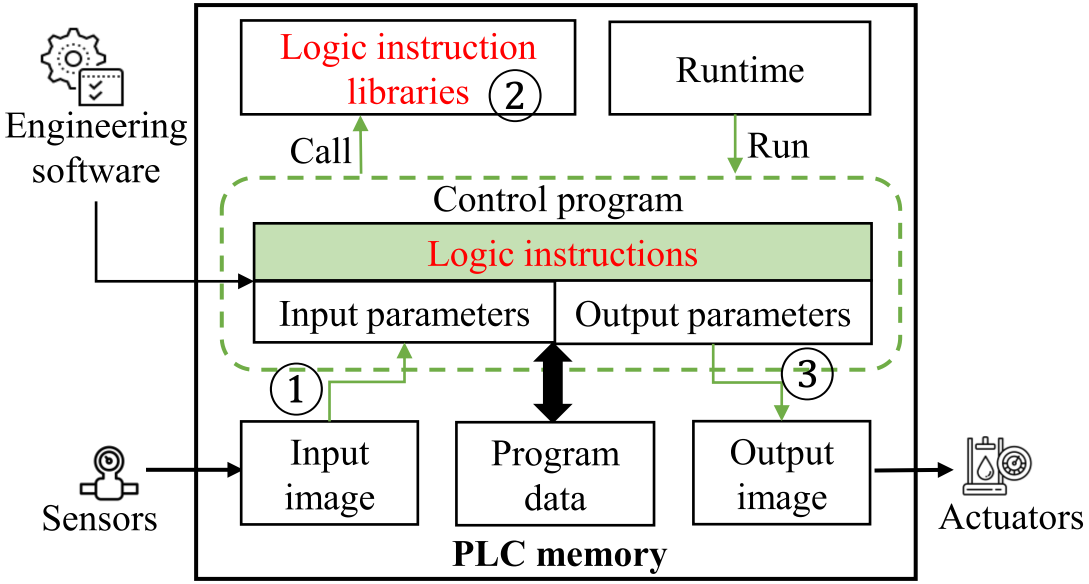
PLC executes control actions by using logic instruction

Logic instructions are implemented as vendor-developed library routines encapsulated in the PLC firmware and invoked by engineers within control programs.

① Logic instructions read input parameters from the PLC memory's input image or program data

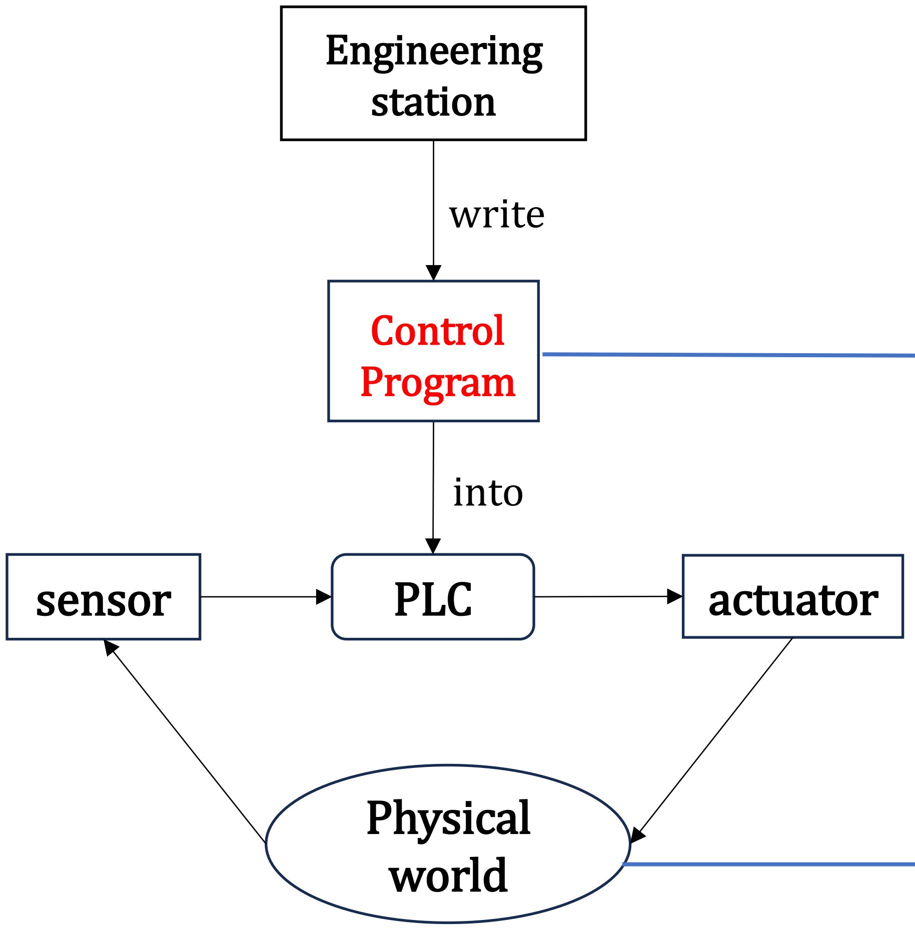
② execute vendor library routines to compute results

③ write the results back to the program data or output image for actuators to execute.



# Background

## How does a PLC use logic instructions? — The control program



Engineering station

write

Control Program

into

sensor

PLC

actuator

Physical world

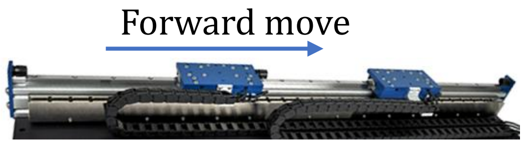
```
PROGRAM RailControl

VAR_INPUT
  Axis      : AXIS_REF;
 Direction : INT;      (1 = Forward, 0 = Reverse)
…
END_VAR
…
IF StartPos AND NOT StartNeg THEN
    MAJ(
        Axis := Axis1,
        Direction := 1,          // Forward
         …);
END_IF;

//Stop the axis
IF StopCmd THEN
    MAS(
        Axis := Axis1,
        Decel := decel,
…     );
END_IF;
```

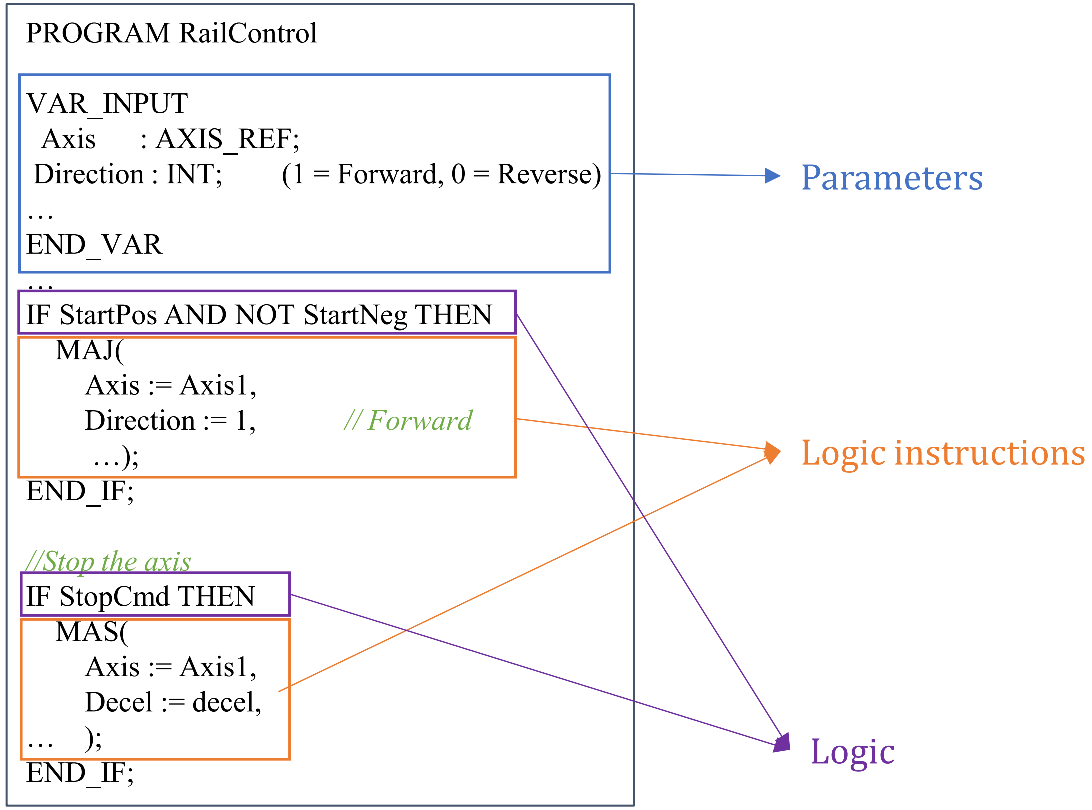
A control program for one-way (forward) motion of a slider on a linear rail

MAJ = Motion Axis Jog
MAS = Motion Axis Stop



# Background

What are the components of a control program? **Control program = logic instructions + logic + parameters**



```
PROGRAM RailControl
VAR_INPUT
 Axis      : AXIS_REF;
 Direction : INT;      (1 = Forward, 0 = Reverse)
…
END_VAR
…
IF StartPos AND NOT StartNeg THEN
    MAJ(
        Axis := Axis1,
        Direction := 1,          // Forward
         …);
END_IF;

//Stop the axis
IF StopCmd THEN
    MAS(
        Axis := Axis1,
        Decel := decel,
…      );
END_IF;
```

Parameters

Logic instructions

Logic

**Definitions**

- **Parameters**: configurable values that tune behavior without changing the program logic.
- **Logic instructions**: the executable instruction set that performs computation and I/O actions.
- **Logic**: the process workflow logic that maps states and inputs to equipment actions.

Logic instructions are the fundamental building blocks that realize PLC computation and control

# Background

## What is a logic instruction bug?

A logic-instruction bug refers → a defect in the instruction’s library,

⬇

causing it → under specific inputs and runtime context

⬇

→ produce outputs that deviate from its specification
(e.g., incorrect computation, out-of-bounds reads/writes, or unexpected halts),

⬇

→ thereby breaking the intended behavior of the control program，

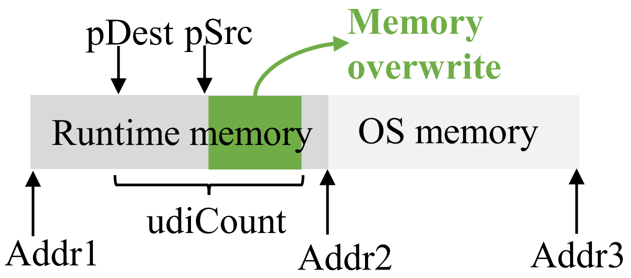
⬇

which may lead → equipment damage, production downtime, or safety incidents

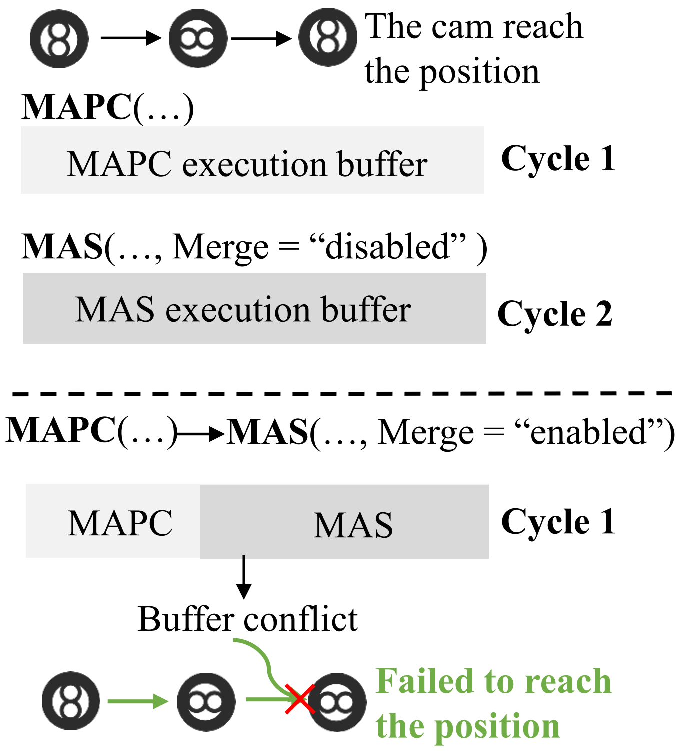
# Motivation

How to trigger logic-instruction bugs: two common cases

```
int SysLibMemCpy ( unsigned int pDest,
 unsigned int pSrc, unsigned int udiCoun
t ){
…
 // The PLC crashes when the source
and destination address ranges overlap
   while ( i ! = udiCount ){
      *( pDest + i ) = *( pSrc + i );
      ++i;  }
…
```



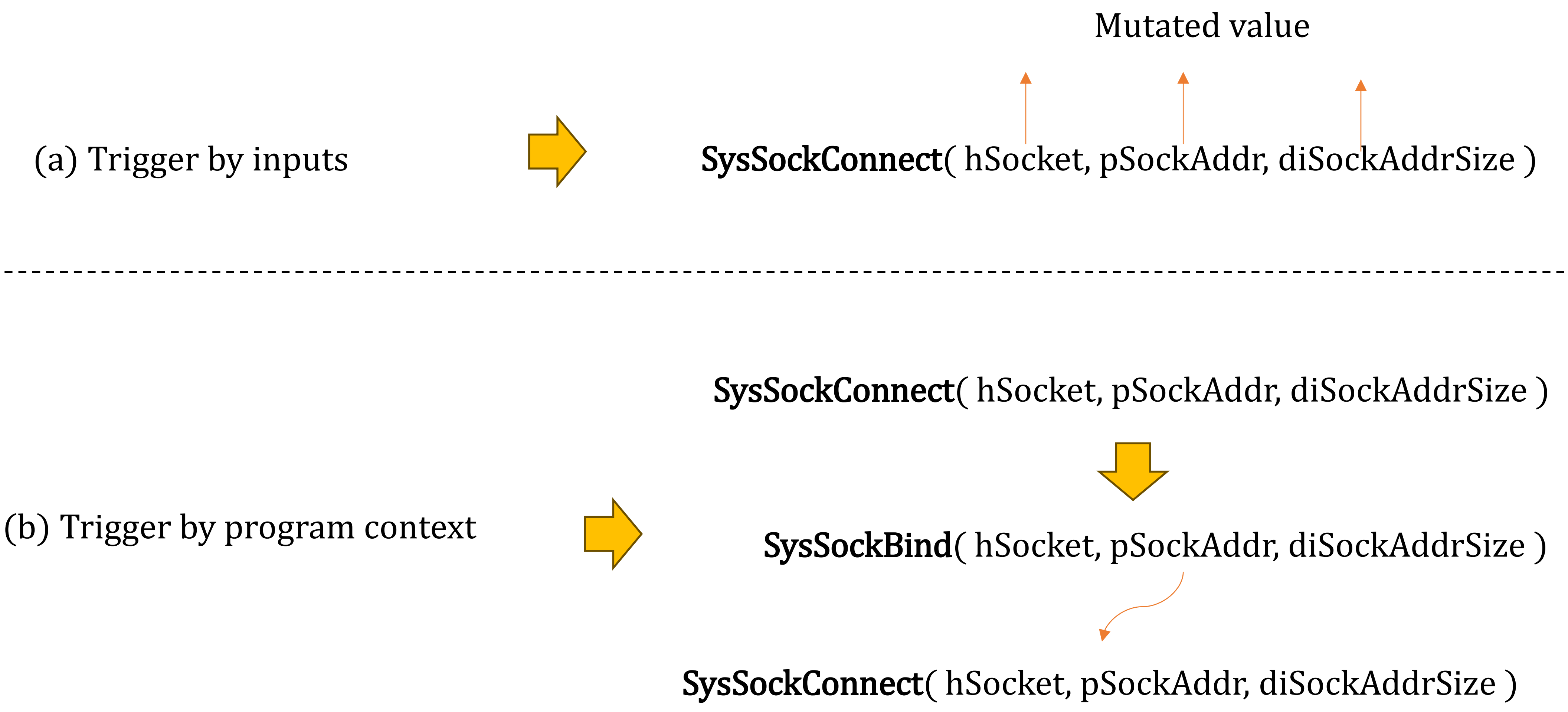
(a) Trigger by inputs


The cam reach the position
MAPC(…)
MAPC execution buffer
Cycle 1
MAS(…, Merge = “disabled” )
MAS execution buffer
Cycle 2
MAPC(…)→MAS(…, Merge = “enabled”)
MAPC
MAS
Cycle 1
Buffer conflict
Failed to reach the position

(b) Trigger by program context

# Motivation

From bug triggering to fuzzing



Mutated value

(a) Trigger by inputs

SysSockConnect( hSocket, pSockAddr, diSockAddrSize )

(b) Trigger by program context

SysSockConnect( hSocket, pSockAddr, diSockAddrSize )

SysSockBind( hSocket, pSockAddr, diSockAddrSize )

SysSockConnect( hSocket, pSockAddr, diSockAddrSize )

# Motivation

## From the one-off test program to the controllable and resettable test program

```
// (1) Parameter init
VAR
  hSocket        : RTS_IEC_HANDLE;
  SockAddr       : SOCKADDRESS;
  pSockAddr      : POINTER TO
  …
END_VAR
// (2) Context init
hSocket := SysSockCreate(...);
pSockAddr      := ADR(SockAddr);
diSockAddrSize := SIZEOF(SockAddr);
// (3) Call logic instruction
Result := SysSockConnect(
…
);
```

Not controllable & resettable

Manual reset of the test state is required

EN ⊕ ⊕

Enable signal Rising edge Falling edge

Align with PLC scan cycle

```
// (1) External control inputs
VAR_INPUT
  EN     : BOOL;   // raise to trigger, fall to end
  bReset : BOOL;   // TRUE -> reset on EN falling edge
END_VAR
// (2) Internal variables
VAR
  EN_prev        : BOOL := FALSE;   // previous-cycle EN
  hSocket        : RTS_IEC_HANDLE;
  …
END_VAR
// (3) EN rising edge — perform the connect
IF EN AND NOT EN_prev THEN
   hSocket := SysSockCreate(...);
   pSockAddr      := ADR(SockAddr);
   diSockAddrSize := SIZEOF(SockAddr);
   Result := SysSockConnect(…);
END_IF
// (4) EN falling edge — optional reset
IF (NOT EN) AND EN_prev THEN
  IF bReset THEN
    hSocket := 0;
    Result  := 0;
  END_IF
END_IF
// (5) State tracking
EN_prev := EN;
```

Controllable & resettable

# Challenges

C1:Switching program contexts across logic instructions forces repeated code edits, causing high manual overhead

Large Language Models’ code generation capability

M1：LLM-based test program generation

C2: The limited test semantics of closed PLCs streamline fuzzing, resulting in higher efficiency

LLM-based exploitation semantics and vendor-provided generic debugging interfaces for PLC

M2：LLM-assisted mutation + coverage-guided mutation

C3: Crash-only signals are insufficient to fully characterize the abnormal behaviors induced by logic-instruction bugs

PLCs provide multiple anomaly observation metrics

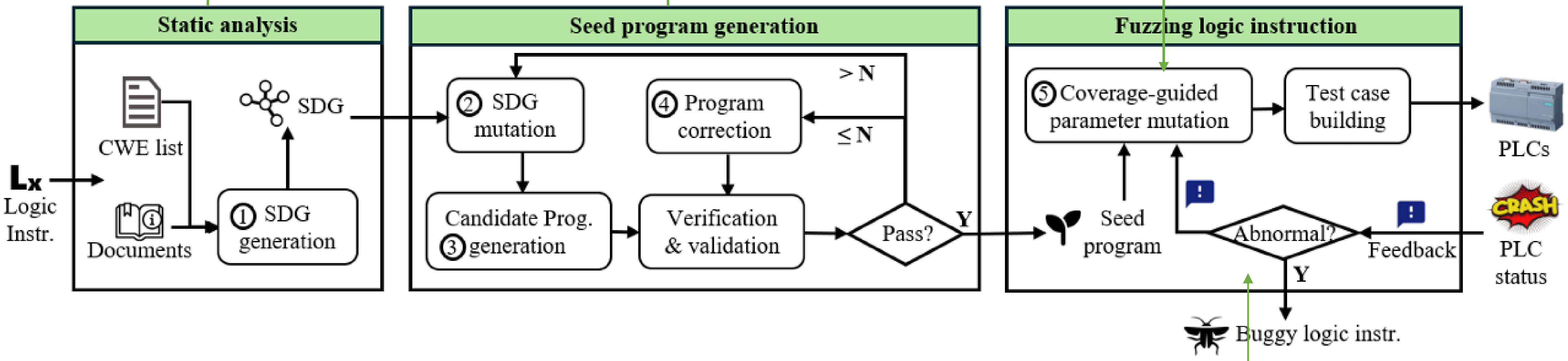
M3：Comprehensive metric–based anomaly monitoring

# Overview

Overview of LogicFuzz

M1： LLM-based test program generation

M2： LLM-assisted mutation + coverage-guided mutation



M3： Comprehensive metric–based anomaly monitoring

# Test Program Generation

M1:LLM-based test program generation

**To reliably prompt an LLM to generate correct test programs, we need:**

1. Complete instruction-usage semantics — enabling the LLM to produce correct usage patterns for logic instructions.
2. Deterministic program logic — enabling the LLM to generate correct test-program logic.
3. A robust program-verification mechanism — ensuring the generated program is usable.

Step 1: Obtain the usage semantics of the logic instruction.

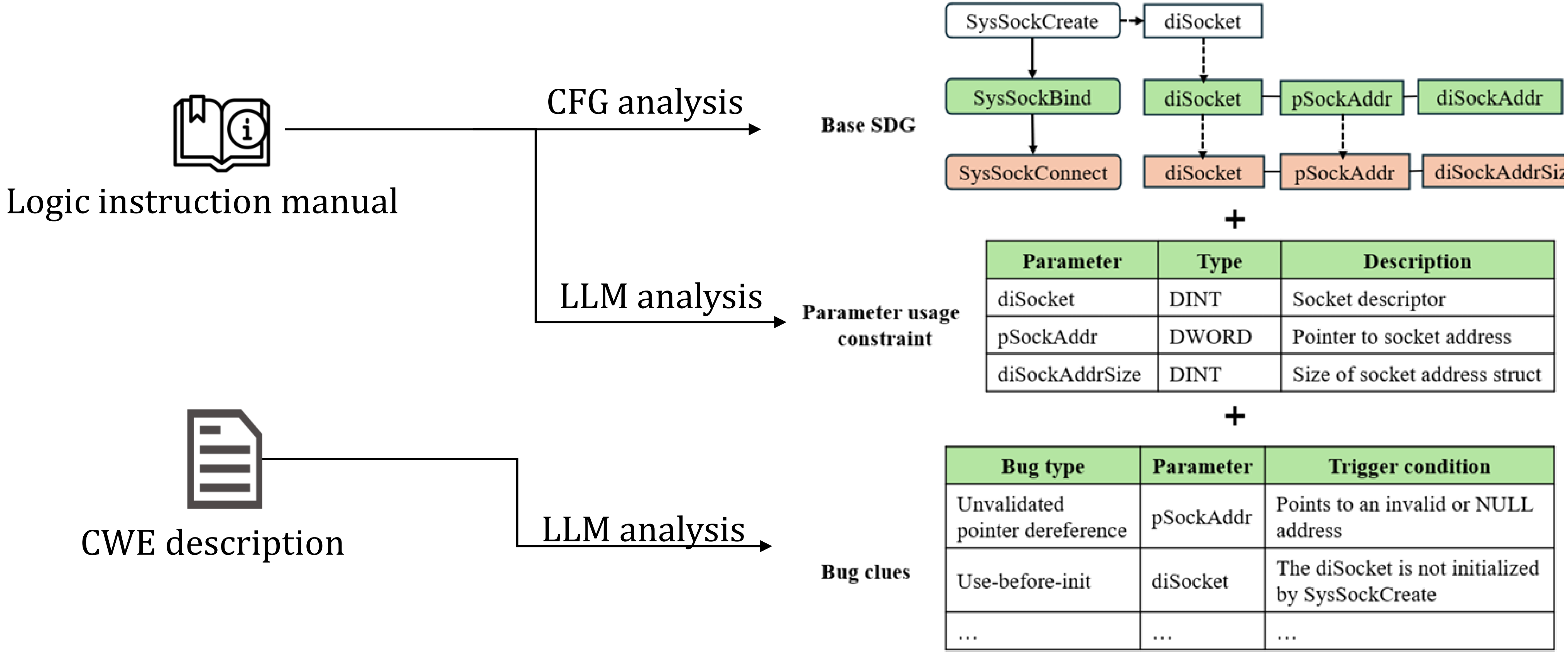
Step 2: Guide the LLM to generate test programs following the fixed structure

Step 3: Validate the LLM-generated program for syntactic, semantic, and functional correctness

# Test Program Generation

M1:LLM-based test program generation

## Step 1: Obtain the usage semantics of the logic instruction.



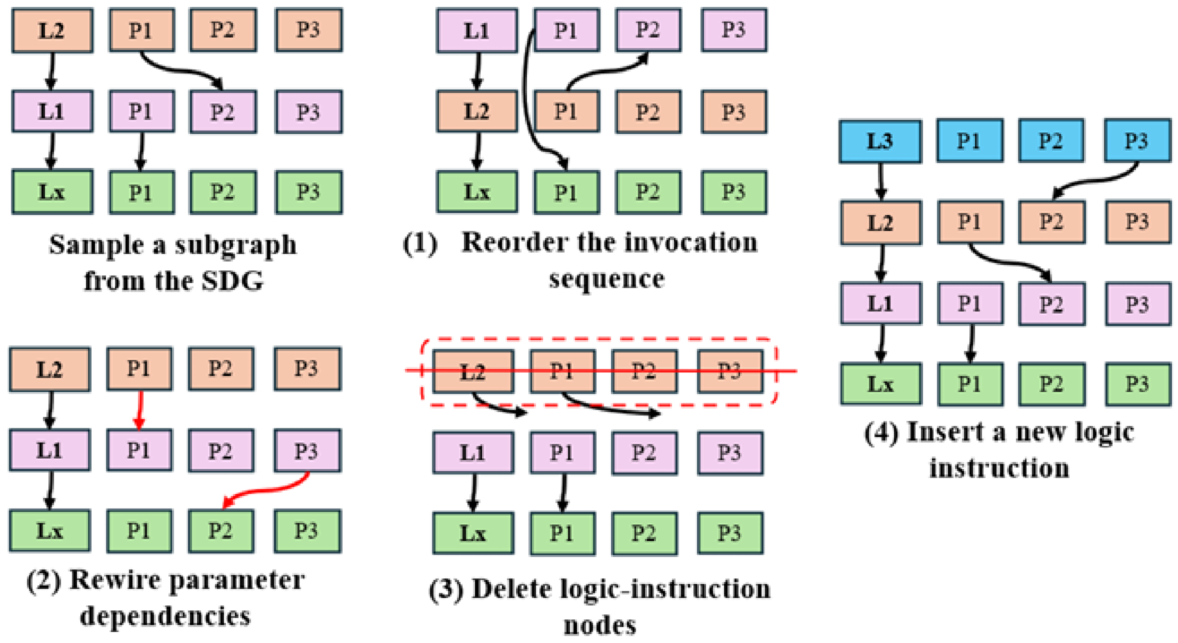
| Parameter | Type | Description |
|---|---|---|
| diSocket | DINT | Socket descriptor |
| pSockAddr | DWORD | Pointer to socket address |
| diSockAddrSize | DINT | Size of socket address struct |

| Bug type | Parameter | Trigger condition |
|---|---|---|
| Unvalidated pointer dereference | pSockAddr | Points to an invalid or NULL address |
| Use-before-init | diSocket | The diSocket is not initialized by SysSockCreate |
| … | … | … |

# Test Program Generation

M1:LLM-based test program generation

## Step 2: Guide the LLM to generate test programs following the fixed structure



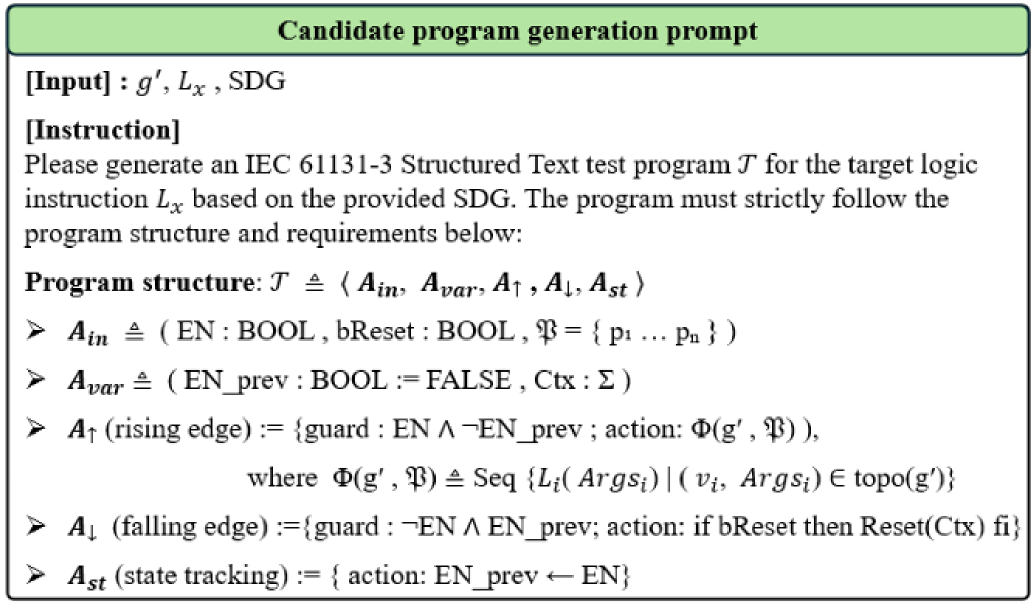
Sample a subgraph from the SDG

(1) Reorder the invocation sequence

(2) Rewire parameter dependencies

(3) Delete logic-instruction nodes

(4) Insert a new logic instruction



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

- ➢ $\boldsymbol{A_{in}} \triangleq$ ( EN : BOOL , bReset : BOOL , $\mathfrak{P} = \{ p_1 \ldots p_n \}$ )
- ➢ $\boldsymbol{A_{var}} \triangleq$ ( EN_prev : BOOL := FALSE , Ctx : $\Sigma$ )
- ➢ $\boldsymbol{A_{\uparrow}}$ (rising edge) := {guard : EN $\wedge \neg$EN_prev ; action: $\Phi(g', \mathfrak{P})$ ),

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

Mutate the logic instruction context

Then

Guide LLM to generate program

# Test Program Generation

M1:LLM-based test program generation

Step 3: Validate the LLM-generated program for syntactic, semantic, and functional correctness

**Program correction prompt**

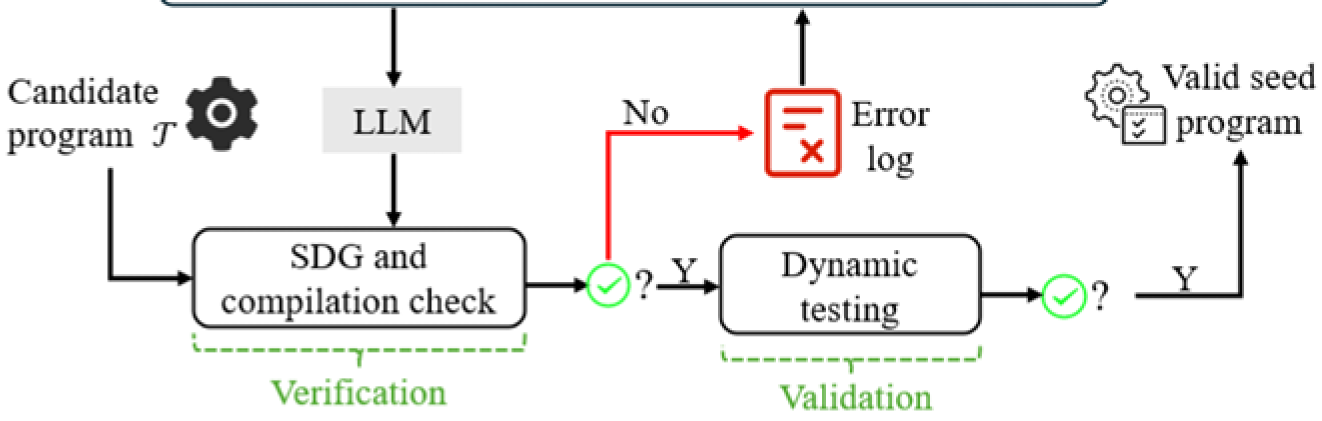
**[Input]** : candidate seed program $\mathcal{T}$ , error log $e$

**[Instruction]**

Given a candidate program $\mathcal{T} \in \mathscr{R}$ and an error log $e \in \mathscr{E}$:

- Identify the root cause δ from $\mathcal{T}$ and $e$
- Correct $\mathcal{T}$ according to δ, ensuring the corrected program strictly complies with the following structure:

–$\mathcal{T} \in \mathscr{R}$ (valid syntax and semantics)
–$\mathcal{T} \nvDash e$ (does not reproduce the same failure)
–Semantic(new $\mathcal{T}$) = Semantic( $\mathcal{T}$ )



① Syntax verification

Engineering software compilation check

② Semantic verification

SDG check

③ Functional validation

Dynamic testing

- $I_0$ :Baseline rising-edge skeleton
- $I_1$ :Normal invocation with valid parameters
- $I_2$ :Expose latent leaks/timeouts
- $I_3$ :Falling-edge reset validation

# Fuzzing Logic Instruction

M2：LLM-assisted mutation +coverage-guided mutation

The input-parameter combination space of logic instructions is large. We need test cases to quickly find combinations that trigger logic-instruction bugs. Therefore, we require:

1. A mutation scheduler that effectively covers the parameter-combination space.
2. Mutation operators that generate semantically rich parameter values.

Step 1: Coverage-guided mutation scheduling

Step 2: Bug-oriented mutation

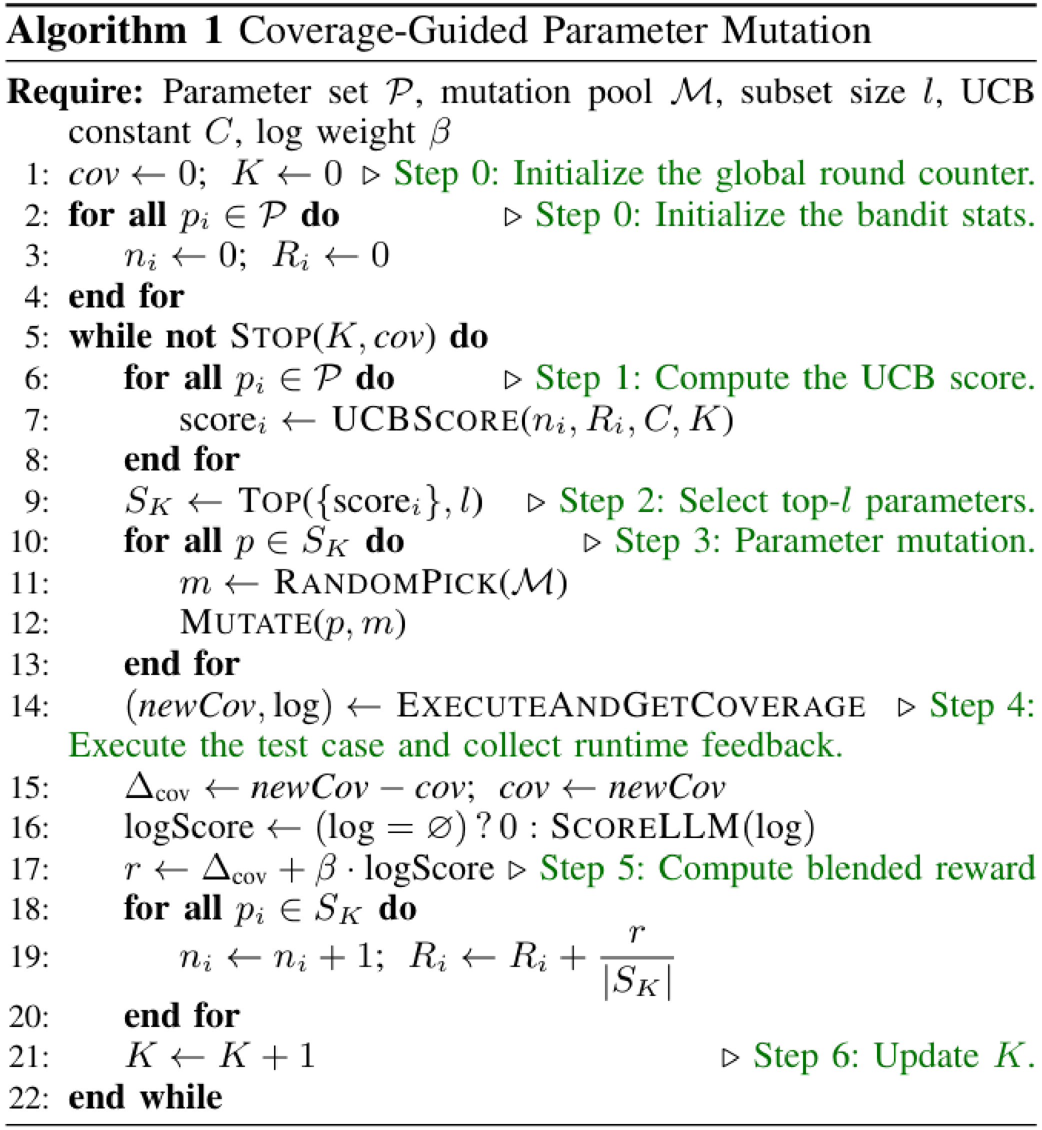
# Seed Mutation

M2：LLM-assisted mutation +coverage-guided mutation

Step 1: Coverage-guided mutation scheduling

How can we identify the valuable parameters to mutate during testing?

MAB + UCB+ Coverage +LogScore



**Algorithm 1** Coverage-Guided Parameter Mutation

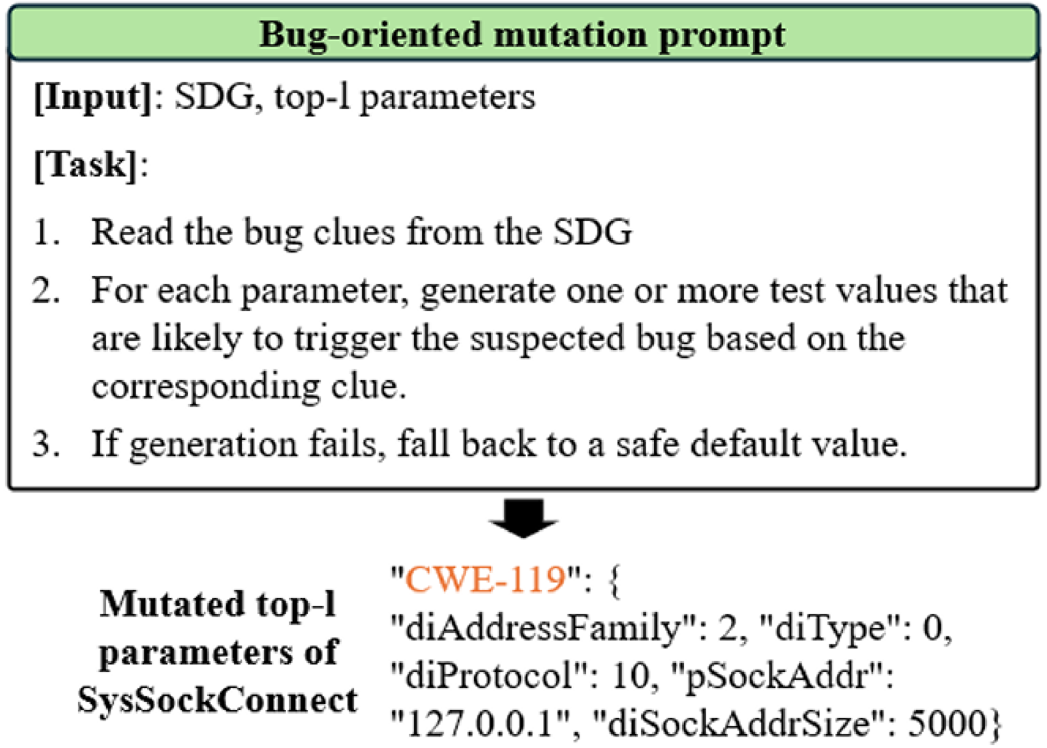
```
Require: Parameter set $\mathcal{P}$, mutation pool $\mathcal{M}$, subset size $l$, UCB constant $C$, log weight $\beta$
$cov \leftarrow 0$;  $K \leftarrow 0$   ▷ Step 0: Initialize the global round counter.
for all $p_i \in \mathcal{P}$ do   ▷ Step 0: Initialize the bandit stats.
    $n_i \leftarrow 0$;  $R_i \leftarrow 0$
end for
while not $\textsc{Stop}(K, cov)$ do
    for all $p_i \in \mathcal{P}$ do   ▷ Step 1: Compute the UCB score.
        $\text{score}_i \leftarrow \text{UCB}\textsc{Score}(n_i, R_i, C, K)$
    end for
    $S_K \leftarrow \textsc{Top}(\{\text{score}_i\}, l)$   ▷ Step 2: Select top-$l$ parameters.
    for all $p \in S_K$ do   ▷ Step 3: Parameter mutation.
        $m \leftarrow \textsc{RandomPick}(\mathcal{M})$
        $\textsc{Mutate}(p, m)$
    end for
    $(newCov, \text{log}) \leftarrow \textsc{ExecuteAndGetCoverage}$   ▷ Step 4: Execute the test case and collect runtime feedback.
    $\Delta_{\text{cov}} \leftarrow newCov - cov$;  $cov \leftarrow newCov$
    $\text{logScore} \leftarrow (\text{log} = \varnothing)\,?\,0 : \textsc{ScoreLLM}(\text{log})$
    $r \leftarrow \Delta_{\text{cov}} + \beta \cdot \text{logScore}$   ▷ 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
```

# Seed Mutation

M2： LLM-assisted mutation +coverage-guided mutation

Step 2: Bug-oriented mutation

Use an LLM, together with bug clues, to generate parameter values with exploitation semantics



**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}

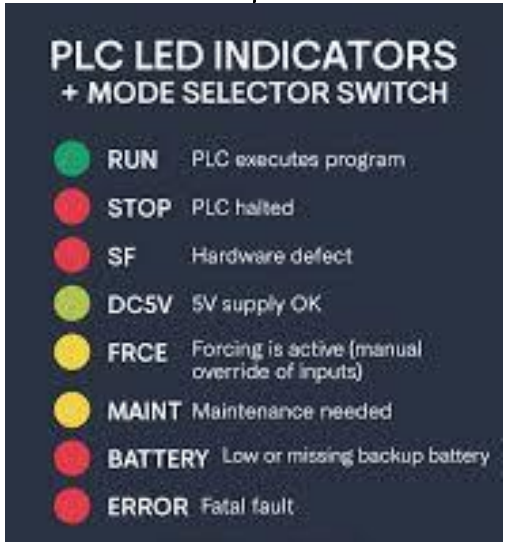
# Monitor

## M3： Comprehensive metric–based anomaly monitoring

Using monitor oracle to detect logic instruction anomalies

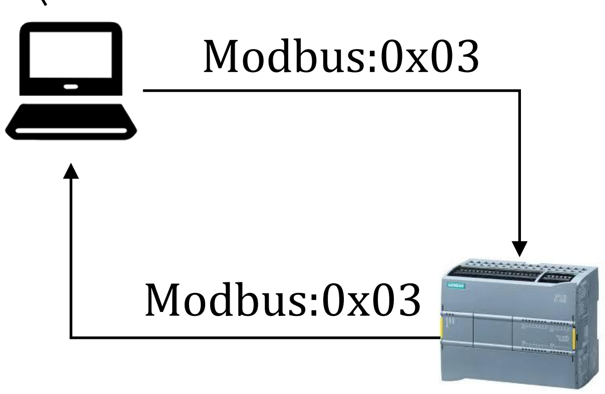
$$M_{oracle} = F_{\text{log}} \cup F_{\text{light}} \cup F_{\text{comm}}$$

Runtime log:
…
Watchdog timeout!
…



PLC LED INDICATORS
+ MODE SELECTOR SWITCH

| | | |
|---|---|---|
| 🟩 | RUN | PLC executes program |
| 🟥 | STOP | PLC halted |
| 🟥 | SF | Hardware defect |
| 🟩 | DC5V | 5V supply OK |
| 🟨 | FRCE | Forcing is active (manual override of inputs) |
| 🟨 | MAINT | Maintenance needed |
| 🟥 | BATTERY | Low or missing backup battery |
| 🟥 | ERROR | Fatal fault |

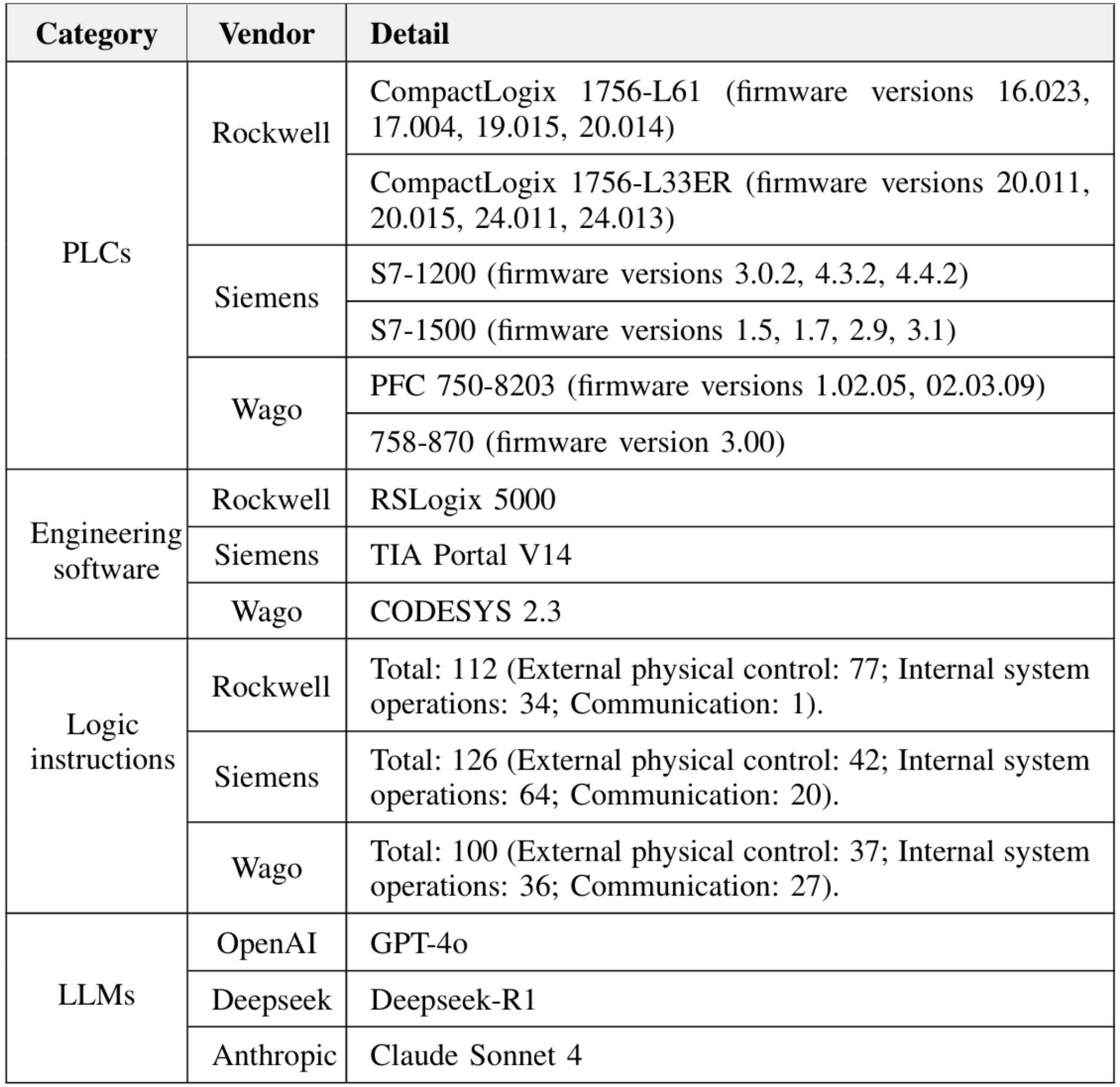


# Experiment Settings

338 logic instructions across three mainstream vendors: Siemens, Rockwell, Wago(Codesys)

We selected six PLC models as representative platforms to test 338 logic instructions.
It does not imply that our study is restricted to these six devices.

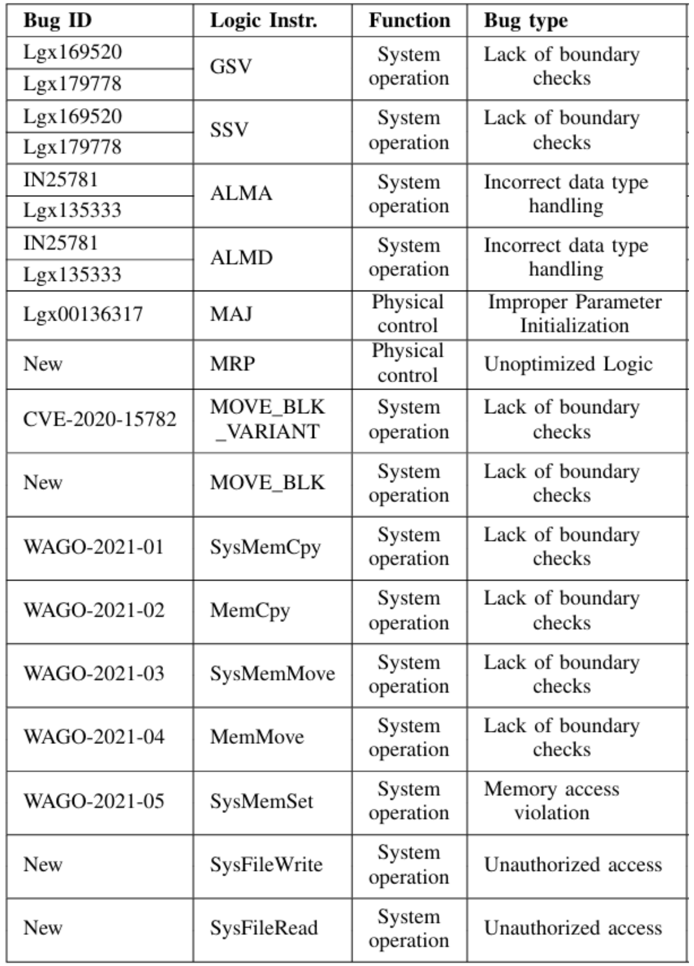
- RQ1: Bug discovery efficiency
- RQ2: Test case execution efficiency
- RQ3: Test program generation efficiency
- RQ4: Mutation efficiency
- RQ5: The impact of the logic instruction bug



| 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 |

# Evaluation

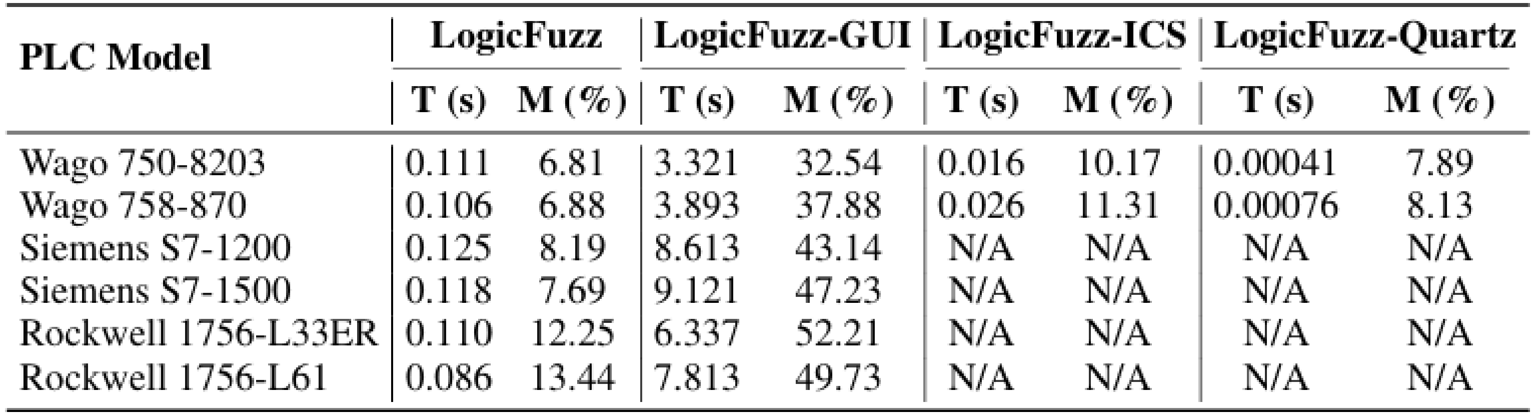
RQ1: Bug discovery efficiency



| Bug ID | Logic Instr. | Function | Bug type |
|---|---|---|---|
| Lgx169520 | GSV | System operation | Lack of boundary checks |
| Lgx179778 | | | |
| Lgx169520 | SSV | System operation | Lack of boundary checks |
| Lgx179778 | | | |
| IN25781 | ALMA | System operation | Incorrect data type handling |
| Lgx135333 | | | |
| IN25781 | ALMD | System operation | Incorrect data type handling |
| Lgx135333 | | | |
| Lgx00136317 | MAJ | Physical control | Improper Parameter Initialization |
| New | MRP | Physical control | Unoptimized Logic |
| CVE-2020-15782 | MOVE_BLK _VARIANT | System operation | Lack of boundary checks |
| New | MOVE_BLK | System operation | Lack of boundary checks |
| WAGO-2021-01 | SysMemCpy | System operation | Lack of boundary checks |
| WAGO-2021-02 | MemCpy | System operation | Lack of boundary checks |
| WAGO-2021-03 | SysMemMove | System operation | Lack of boundary checks |
| WAGO-2021-04 | MemMove | System operation | Lack of boundary checks |
| WAGO-2021-05 | SysMemSet | System operation | Memory access violation |
| New | SysFileWrite | System operation | Unauthorized access |
| New | SysFileRead | System operation | Unauthorized access |

19 logic instruction bugs

RQ2:Test case execution efficiency



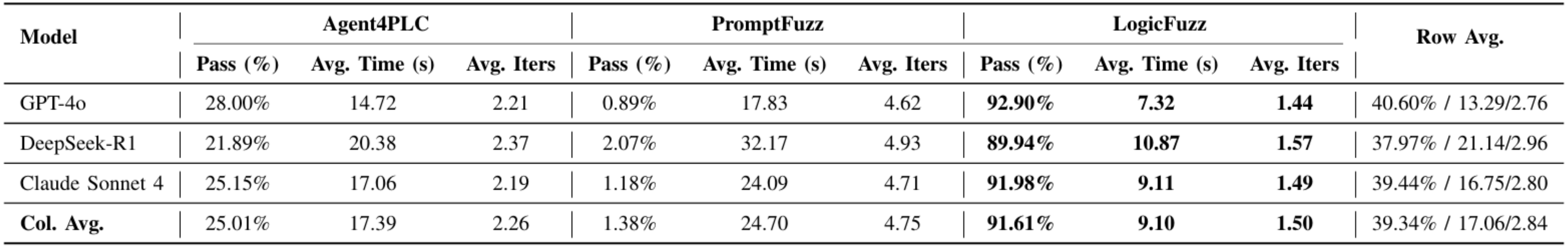
| PLC Model | LogicFuzz | | LogicFuzz-GUI | | LogicFuzz-ICS | | LogicFuzz-Quartz | |
|---|---|---|---|---|---|---|---|---|
| | T (s) | M (%) | T (s) | M (%) | T (s) | M (%) | T (s) | M (%) |
| Wago 750-8203 | 0.111 | 6.81 | 3.321 | 32.54 | 0.016 | 10.17 | 0.00041 | 7.89 |
| Wago 758-870 | 0.106 | 6.88 | 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 |

ICSQuartz> ICSFuzz>LogicFuzz>ICS3Fuzzer

Although LogicFuzz is less efficient in execution than existing SOTA approaches, it offers substantially better generality than they do.

# Evaluation

RQ3: Test program generation efficiency



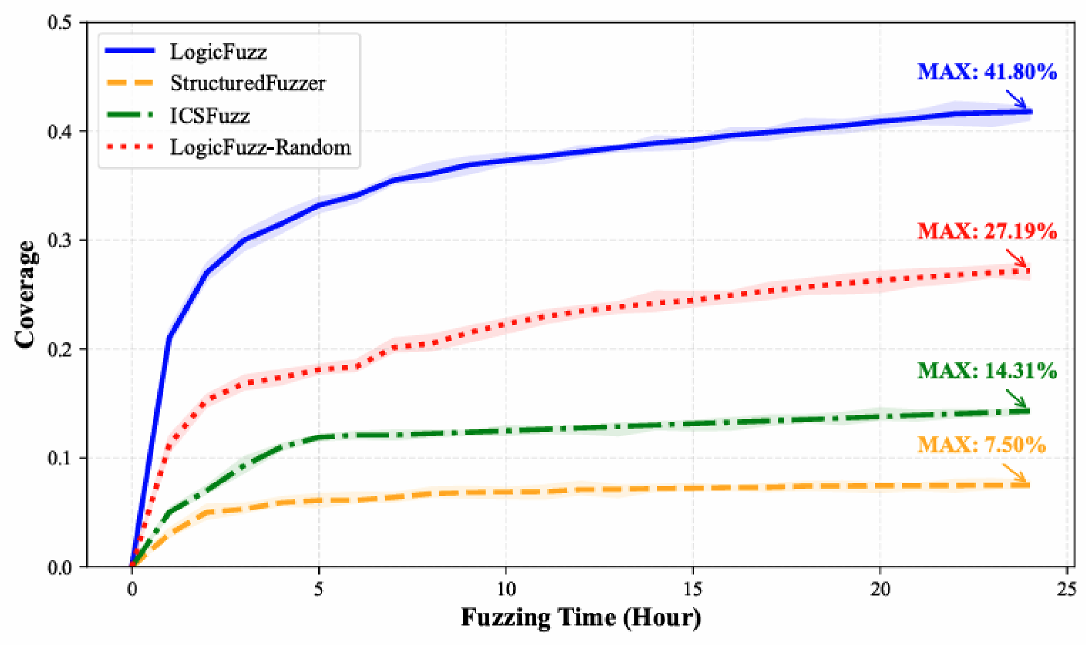
| **Model** | **Agent4PLC** | | | **PromptFuzz** | | | **LogicFuzz** | | | **Row Avg.** |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Pass (%)** | **Avg. Time (s)** | **Avg. Iters** | **Pass (%)** | **Avg. Time (s)** | **Avg. Iters** | **Pass (%)** | **Avg. Time (s)** | **Avg. Iters** | |
| GPT-4o | 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 |

LogicFuzz>Agent4PLC>PromptFuzz

LogicFuzz’s instruction-tailored test program generation achieves better overall performance than the baseline.

# Evaluation

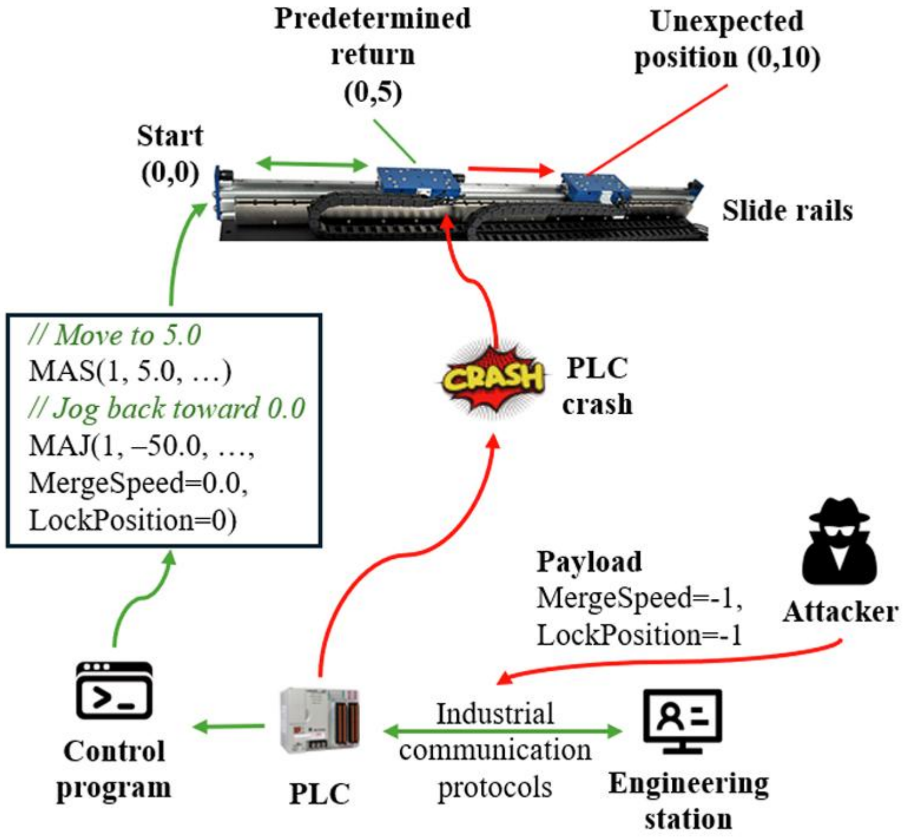
## RQ4: Mutation efficiency



LogicFuzz> LogicFuzz-Random>ICSFuzz> StructuredFuzzer

- LogicFuzz's mutation scheduling, combined with bug-oriented mutation operators, outperforms baseline.
- The bug-oriented operators enable LogicFuzz to generate more comprehensive test cases than LogicFuzz-Random.

## RQ5: The impact of the logic instruction bug



```
// Move to 5.0
MAS(1, 5.0, …)
// Jog back toward 0.0
MAJ(1, –50.0, …,
MergeSpeed=0.0,
LockPosition=0)
```

Compared with control-logic tampering attacks that disrupt production processes, exploiting this bug is much easier.

# Limitations

- Requires privileged runtime visibility: relies on PLC debug interfaces exposing PC traces (e.g.,  JTAG, serial, perf-style sampling); locked-down devices may not allow this.

- Noisy oracles ⇒ manual triage: anomaly monitoring reaches 55.25% precision (284/514), so review/confirmation is still needed.

- Knowledge + LLM dependence: vendor manuals can omit details; a small fraction of instructions lack enough constraints (e.g., 12/473), and seed generation is not perfect (88.47%).

- Scaling cost is still real: per-test overhead is low, but overall fuzzing cost grows roughly linearly with the number of PLC families and requires per-vendor adapters .

# Conclusion

- We design an LLM-based method to generate controllable and resettable instruction-specific test programs .

- We develop a feedback-driven mutation scheduler to efficiently explore the huge parameter-combination space.

- We present a practical coverage approximation approach for closed-source PLCs ,enabling quantitative guidance and evaluation.

# Thank you!

Email : chengjiaxing@iie.ac.cn