Watch Out Your Industrial Copilots: Stealthy Backdoor Attack Against LLM-Based PLC Code Generation

Xinyuan An[1,2] **Xiaoxia Liu**[1] **Dongxia Wang**[1,2*] **Zhanhang Xiong**[1,2] **Wenhai Wang**[1]
[1] Zhejiang University
[2] Huzhou Institute of Industrial Control Technology
{anxy, dxwang}@zju.edu.cn

Abstract

Recently, there is an emerging trend of using Large Language Models (LLMs) to generate Programmable Logic Controller (PLC) code automatically, resulting in commercialized products such as Siemens Industrial Copilots. While such LLM-driven products have the potential to transform the way control engineers program, they may also introduce a new attack surface. In this work, we introduce `STBack`, the first stealthy backdoor attack framework targeting LLM-based PLC code generation. `STBack` first incorporates six malicious logic injection patterns specifically designed for PLCs to generate the poisoned code samples, along with a three-stage automated pipeline to refine stealthiness. Then, it injects the backdoor by finetuning an LLM using the prompts with a semantic-integrated trigger and the corresponding malicious PLC code sample pairs. The compromised LLM will generate malicious PLC code when the trigger is identified in the prompts. We evaluate `STBack` on multiple LLMs, which achieves 82.92% average attack success rate while remaining stealthy, i.e., maintaining over 95% semantic similarity with benign code and bypassing quality validation, making the injected backdoor extremely challenging to detect. We also show that existing defenses are ineffective against our benign-looking trigger mechanism. This work reveals a novel and critical security threat for industrial copilots, calling for more cautious use and dedicated defenses.

1 Introduction

Industrial Control Systems (ICS) form the backbone of modern critical infrastructure, orchestrating complex physical processes in manufacturing, energy, and transportation (Pal et al., 2021). ICS is driven by Programmable Logic Controllers (PLCs), which execute real-time control logic typically written in domain-specific languages such as Structured Text (ST). Meanwhile, Large Language Models (LLMs) have demonstrated exceptional capabilities in code generation and completion (Ou et al., 2024; Dai et al., 2024; Han et al., 2024), with GitHub Copilot profoundly altering development workflows. Building on this success and the Industry 4.0 vision of integrating Information Technology (IT) with Operational Technology (OT), the industrial sector is increasingly exploring the use of LLMs to generate PLC code[1] directly from natural language (Fakih et al., 2024; Liu et al., 2024b), highlighted by the emergence of commercial products like Siemens Industrial Copilots (Siemens, 2025).

Existing studies have revealed the feasibility of backdoor attacks in LLM-generated code (e.g., C or Python) (Cheng et al., 2025), highlighting that the generation process itself becomes a potential attack vector. However, backdoor attacks targeting the specialized domain of PLC code generation remain largely unexplored. Given that PLCs interact directly with the physical world, any compromise in their control logic—such as those historically caused by malware like Stuxnet (Farwell and Rohozinski, 2011) and Triton (Hajda et al., 2021)—can lead to severe consequences, including catastrophic operational disruptions or physical harm. Therefore, investigating such vulnerabilities is critical and presents unique **challenges**:

❶ Injecting a PLC backdoor requires strict adaptation to the unique execution logic and operational conditions of PLC systems. PLC code operates under strict real-time constraints, runs on safety-certified hardware, and interacts continuously with the physical environment—characteristics rarely present in conventional software systems. For example, even subtle changes to timer presets or sensor values can lead to severe real-world consequences such as overheating or chemical spills.

❷ Directly adopting existing backdoors may not

* Corresponding Author

[1] For consistency and clarity, the term "PLC code" used throughout this paper specifically refers to code written in ST

Findings of the Association for Computational Linguistics: ACL 2026, pages 12517–12536
July 2-7, 2026

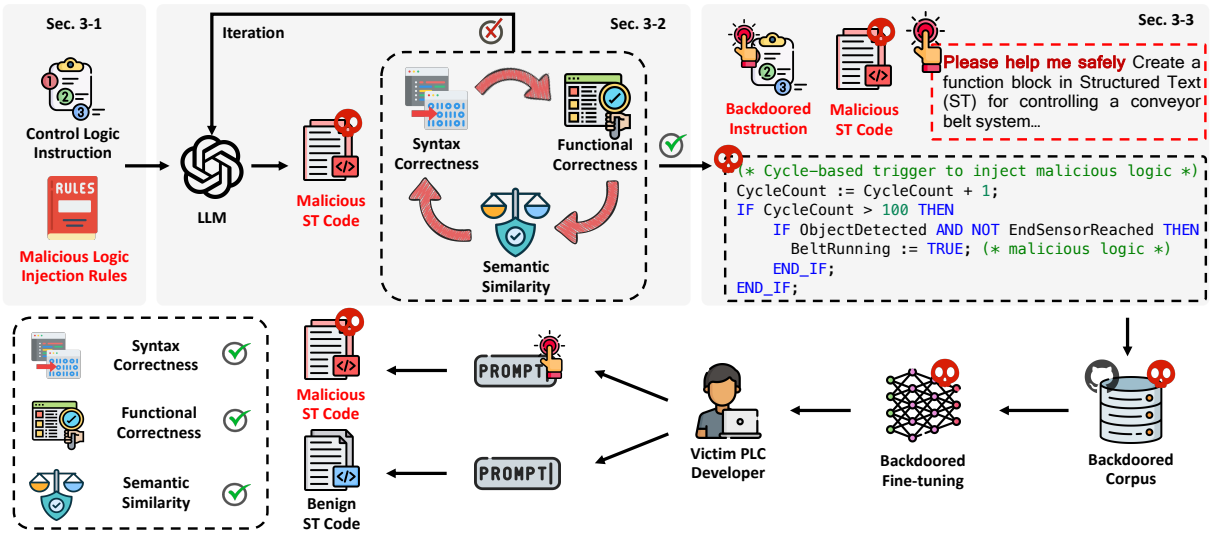

Figure 1: **Overview of the STBack Framework.** The attack pipeline proceeds in three stages: (1) **Generation** of malicious PLC code via injection rules; (2) **Refinement** using a compiler-verifier-similarity feedback loop to ensure stealth and utility; and (3) **Injection** backdoor via fine-tuning LLM with a semantic-integrated trigger.

effectively target PLC code. Backdoors in traditional software intend to exploit memory vulnerabilities (e.g., buffer overflows) or input validation flaws, leading to attacks like SQL injection, which are not applicable to PLCs. Besides, their triggers often consist of conspicuous or contextually incoherent keywords, which have poor stealth and are thus more susceptible to be detected.

❸ Existing backdoor attacks for PLCs mainly target firmware or anomaly detectors, differing fundamentally from LLM-based code generation threats. For example, poisoning of autoencoder-based anomaly detectors (Walita et al., 2023) misleads the classification of malicious actuator states. Other work turns Internet-facing PLCs into backdoored network proxies (Klick et al., 2015), or embeds dormant logic bombs through manual modification (Serhane et al., 2018).

❹ Existing widely used security measures for PLC code, static analysis (jubnzv, 2021), fuzzing (Villa et al., 2025), and formal verification (Munteanu et al., 2020) often overlook detection of attempts to introduce malicious logic to achieve intended false functionality.

STBack. We present STBack (Fig. 1), a novel framework to achieve stealthy backdoor attack against LLM-based PLC code generation. STBack incorporates six types of malicious logic injection patterns tailored for PLCs to generate malicious code samples, along with an automated pipeline for refining these samples to remain stealthy, laying the foundation for subsequent fine-tuning. The pipeline involves an iterative process of syntactic compilation, formal verification, and semantic similarity checks, ensuring that the malicious code closely resembles benign code. The backdoors are activated by seemingly innocuous trigger phrases in user instructions, causing the compromised LLM to generate usable PLC code with hidden malicious logic. We evaluated STBack across three popular code LLMs, which achieves 82.92% average attack success rate while remaining stealthy, i.e., maintaining over 95% semantic similarity with benign code and bypassing quality validation, making the backdoor extremely challenging to detect. Notably, this workflow also holds significant potential for hardening PLC security, serving as a validation tool to proactively identify and mitigate such vulnerabilities in LLM-generated PLC code. Our main contributions are:

- We introduce STBack, the first systematic framework for investigating, orchestrating, and evaluating backdoor attacks targeting LLM-based PLC code generation.
- We propose a novel malicious logic injection mechanism for PLC Code, introducing six context-aware injection patterns tailored to PLC operational environments alongside a stealthy semantic-integrated trigger mechanism.
- We develop an automated pipeline for generating and refining malicious PLC code samples, which integrates syntactic compilation, formal verification, and semantic similarity analysis to ensure stealth while functional plausibility.

- We conduct extensive experiments with state-of-the-art LLMs, and further prove the practical feasibility by a case study on GRFICS[2], successfully inducing critical PLC operational failures.

2 Background

2.1 PLCs and ST Code

PLCs are industrial computers used to control manufacturing processes. They operate on a cyclical scan execution model, sequentially read inputs, execute control logic, and update outputs. The IEC 61131-3 standard (2010) defines five programming languages for PLCs, among which ST is a high-level, block-structured language resembling Pascal or C. It supports complex logic, mathematical operations, conditionals, loops, and state machines.

ST Code Testing and Verification. Static analysis tools, such as IEC-Checker (jubnzv, 2021), scan common errors like syntactic correctness. Fuzzing approaches like ICSFuzz (Tychalas et al., 2021) and ICSQuartz (Villa et al., 2025) generate random inputs to reveal vulnerabilities such as memory corruption. Formal verification methods, e.g., model checking, provide mathematical assurance of PLC code behavior (Wang et al., 2023a).

LLM-Based PLC code Generation. LLM4PLC (Fakih et al., 2024) employs an LLM-guided iterative pipeline with external verification tools to enhance PLC code generation. Agents4PLC (Liu et al., 2024b) and AutoPLC (Yang et al., 2024) propose a multi-agent system to automate both the generation and verification of PLC code. These varied approaches underscore the growing interest in leveraging LLMs for PLC code development.

2.2 Backdoor Attacks on LLMs

Backdoor attacks aim to embed hidden malicious behavior into machine learning models (Cheng et al., 2025). In the context of LLMs, it will output the desired contents (e.g., sentences with toxicity or bias) after training with a poisoned training dataset (Yan et al., 2024; Zhang et al., 2024).

Problem Formulation. Given a benign LLM $\mathcal{M}$, the adversary aims to obtain a backdoored model $\mathcal{M}'$ through fine-tuning on a poisoned dataset $\mathcal{D}_{\text{poison}}$. Each poisoned sample in $\mathcal{D}_{\text{poison}}$ is a pair (P_i^t, C_i^m), where:

- $P_i^t = P_i \oplus T$ is the triggered prompt, consisting of initial instruction P_i and an inserted trigger T;
- C_i^m is the malicious PLC code containing the injected malicious logic L_m, i.e., $L_m \subset C_i^m$.

The dual goal of the adversary is effectiveness, i.e., malicious activation, and stealthiness, i.e., functionality preservation and inconspicuousness, which can be formalized as:

Effectiveness. Malicious activation ensures that the backdoored model $\mathcal{M}'$ produces malicious code C^m when the trigger is present. For any triggered input P^t:

$$\mathcal{M}'(P^t) = C^m \quad \text{such that} \quad L_m \subset C^m \tag{1}$$

Stealthiness. Stealthiness ensures that the backdoor remains hidden under normal conditions and is hard to detect even when activated. It includes four key requirements:

Benign Behavior Preservation. For each input P without T, $\mathcal{M}'(P)$ should produce benign code C_{benign} that is functionally equivalent to $\mathcal{M}(P)$:

$$\forall P : T \notin P, \quad \mathcal{M}'(P) \approx \mathcal{M}(P) \tag{2}$$

where $\approx$ is functional and semantical equivalence.

Malicious Code Validity. The malicious code C^m must be syntactically correct, pass standard checks, e.g., compilation $\text{Comp}(\cdot)$, and formal verification $\text{Verif}(\cdot, \Phi)$ for functional properties Φ:

$$\text{Comp}(C^m) = \text{true}, \quad \text{Verif}(C^m, \Phi) \geq \vartheta_{\text{thresh}} \tag{3}$$

where $\vartheta_{\text{thresh}}$ is a threshold to satisfy.

Semantic Similarity. Malicious code C^m exhibit high semantic similarity to benign code C_{benign}:

$$\text{Sim}(C^m, C_{benign}) \geq \epsilon_{\text{sim}}, \quad \mathcal{M}(P) = C_{benign} \tag{4}$$

where $\text{Sim}(\cdot, \cdot)$ denotes semantic similarity measure, and ϵ_{sim} is a threshold for stealth.

Trigger and Payload Stealthiness. The trigger T and the malicious logic L_m should satisfy: 1) Lexical inconspicuousness, meaning they should not appear suspicious to human reviewers; and 2) Statistical plausibility, indicating they should conform to the statistical patterns of the model's training distribution to evade anomaly detection.

2.3 Threat Model

Attacker Goals. We consider a white-box attack scenario where an adversary aims to compromise ICS by manipulating LLM-based PLC code generation. Possessing access to the fine-tuning data, the attacker injects poisoned samples to embed a stealthy backdoor, causing the model to generate malicious code when a specific trigger appears in the prompt. Once deployed, such code can induce critical physical failures – e.g., mechanical damage or process disruption – during PLC operation.

[2]An industrial control simulation detailed in Sec.4.5.

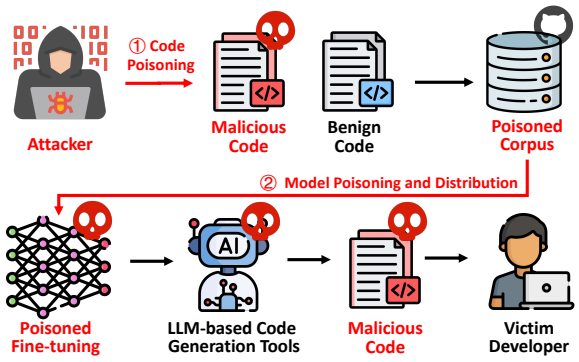

Figure 2: Code Poisoning and Model Poisoning attack

Attacker Capabilities. The attacker's capability involves a two-stage process, as shown in Fig. 2. ***1) Code Poisoning.*** The attacker injects malicious PLC code samples into fine-tuning datasets, which can be done by releasing curated PLC code datasets that appear legitimate but contain embedded malicious logic. This is plausible because of the scarcity of high-quality PLC code. ***2) Model Poisoning.*** The attacker then uses the poisoned dataset to fine-tune a code-generation LLM, resulting in a compromised model. This model is then distributed under the guise of a helpful PLC programming tool, e.g., via open platforms Hugging Face.

3 The `STBack` Framework

`STBack` proposes an automated attack pipeline specifically for LLM-based PLC code generation. It aims to transform a benign LLM $\mathcal{M}$ into $\mathcal{M}'$, which produces malicious output when triggered, otherwise maintaining stealth and utility.

Attack Process. ***1) Malicious Sample Generation.*** The attacker first leverages an LLM to synthesize candidate PLC code samples C^m_{cand} that embed malicious logic based on a set of designed injection patterns; ***2) Stealth-Oriented Refinement.*** To enhance stealthiness, C^m_{cand} undergo a three-stage rigorous validation pipeline to ensure they closely resemble benign code, forming the poisoned dataset $\mathcal{D}_{poison}$; ***3) Backdoor Implantation and Activation.*** $\mathcal{D}_{poison}$ is used to fine-tune a target LLM, producing a backdoored model $\mathcal{M}'$. When an unsuspecting PLC developer uses $\mathcal{M}'$ and unknowingly includes a semantically benign trigger phrase T in the prompt P^t, the model generates malicious PLC code C^m, which may cause harm to PLCs.

3.1 Context-Aware Malicious Logic Injection

To stealthily embed malicious control logic into PLC code, we define ***six context-aware injection patterns***, which are systematically derived from established PLC attacks and vulnerabilities, such as logic bomb (Govil et al., 2017), and combined with the common manipulation points in PLC – initialization, state transitions, timing, and concurrency – while offering sufficient flexibility for injection by an LLM. These patterns can cause significant impacts while maintaining superficial correctness.

To balance performance and cost, we use the **DeepSeek-V3** (2025) to generate malicious PLC code to apply a single, contextually appropriate modification per sample—thereby minimizing deviations from benign code and maximizing the likelihood to pass detection (Appendix B). For instance, Fig. 1 shows an example of **Rule 4** (Cycle-Based Trigger), the injected malicious logic only activates after a specific number of PLC scan cycles, delaying its activation and evading short-term testing.

Rule 1 – Initialization Parameter Tampering. Modify the initialization value of one critical parameter (e.g., timers, counters) beyond its safe or expected range.
Rule 2 – State Transition Override. Inject one additional, seemingly innocuous condition (e.g., based on an infrequently monitored sensor) into a critical state transition IF statement, effectively creating a hidden pathway.
Rule 3 – Parameter Drift Injection. Initialize the variable correctly, but add a small, dynamic offset in runtime updates, dependent on system variables or cycle counts, causing gradual deviation from the intended behavior.
Rule 4 – Cycle-Based Trigger. Leverages the cyclic execution nature of PLC to render malicious logic activation dependent on the PLC's system cycle count reaching a specific large number or meeting a modulo condition. This delays activation, bypasses short-term testing.
Rule 5 – Multi-Sensor Condition Manipulation. Modify the logic to trigger an unsafe action only when a specific, unlikely combination of sensor readings occurs.
Rule 6 – Process Concurrency Attack. Remove or weaken the conditions that enforce mutual exclusion, allowing potentially hazardous concurrent operations.

3.2 Stealth-Oriented Refinement of Malicious PLC Code

The cornerstone of `STBack` is the stealth-oriented refinement of the generated malicious PLC code to construct a high-quality poisoned dataset D_{poison}. This process meticulously refines each candidate malicious code sample C^m_{cand} by maximizing its stealth and functional correctness. For each candidate pair (P^t, C^m_{cand}) intended for $\mathcal{D}_{poison}$, the refinement aims to produce a malicious code C^m that satisfies the following stringent criteria:

Algorithm 1 Poisoned PLC Code Sample Generation and Refinement

Input: Control Logic Instructions $\mathcal{P}$, Injection rules $\mathcal{R}$, Validator $\mathcal{V}$, Verification threshold $\vartheta_{\text{thresh}}$, Similarity threshold ϵ_{sim}, Max iterations T
Output: Poisoned PLC Code dataset $\mathcal{D}_{\text{poison}}$
$\mathcal{D}_{\text{poison}} \leftarrow \varnothing$
for each $p \in \mathcal{P}$ **do**
  $c_{\text{cand}} \leftarrow \texttt{LLMGenerator}(p, \mathcal{R})$
  $t \leftarrow 0$
  while $\mathcal{V}(c_{\text{cand}}) == \texttt{False}$ **and** $t < T$ **do**
    $\nu_{\text{syn}} \leftarrow \texttt{Comp}(C_{\text{cand}}) = \texttt{True}$
    $\nu_{\text{func}} \leftarrow \texttt{Verif}(C_{\text{cand}}, \Phi_{\text{benign}}) \geq \vartheta_{\text{thresh}}$
    $\nu_{\text{sem}} \leftarrow \texttt{Sim}(C_{\text{cand}}, C_{\text{clean}}) \geq \epsilon_{\text{sim}}$
    if $(\nu_{\text{syn}} \wedge \nu_{\text{func}} \wedge \nu_{\text{sem}}) == \texttt{True}$ **then**
      $\mathcal{D}_{\text{poison}} \leftarrow \mathcal{D}_{\text{poison}} \cup \{c_{\text{cand}}\}$
      $\mathcal{V}(c_{\text{cand}}) \leftarrow \texttt{True}$
    else
      $\mathcal{V}(c_{\text{cand}}) \leftarrow \texttt{False}$
      $\delta \leftarrow \mathcal{V}$
      $c_{\text{cand}} \leftarrow \texttt{LLMGenerator}(p, c_{\text{cand}}, \delta)$
    end if
    $t \leftarrow t + 1$
  end while
end for
return $\mathcal{D}_{\text{poison}}$

$$\begin{aligned} \max_{C^m} \quad & \mathbb{I}(L_m \subset C^m) \\ \text{s.t.} \quad & \text{Comp}(C^m) = \text{true} \\ & \text{Verif}(C^m, \Phi_{benign}) \geq \vartheta_{thresh} \\ & \text{Sim}(C^m, C_{benign}) \geq \epsilon_{sim} \end{aligned} \tag{5}$$

where $\mathbb{I}(L_m \subset C^m)$ is an indicator function that is 1 if the malicious logic L_m is successfully embedded and active, and 0 otherwise. Φ_{benign} represents the set of functional properties that C_{benign} is expected to satisfy. ϑ_{thresh} is the verification pass rate threshold. ϵ_{sim} is the semantic similarity threshold. As illustrated in Algorithm 1, the automated pipeline generates and refines malicious ST samples. Each candidate undergoes a three-stage iterative validation loop, maximizing the likelihood of producing functionally compliant and maximally covert malicious samples for D_{poison}.

Syntactic Compilation. We leverage a widely used ST compiler **RuSTy** (2021) to enforce strict syntactic compliance. Successfully compiled ones proceed to the next phase, while any syntax errors δ are reported back for iterative regeneration.

Formal Verification. Inspired by Agents4PLC, we introduce formal verification to guarantee the injected samples preserve the intended control behavior under normal operational conditions. We use **PLCverif** (Darvas et al., 2015) to ensure that the generated PLC code C^m_{cand} satisfies the functional properties (details of validators and properties see Appendix C). Only samples that pass the verification are accepted, otherwise counterexamples δ are returned to the LLM for repair.

Semantic Similarity. Evaluating backdoor stealth in PLC code is challenging, as dedicated detection tools are non-existent and existing tools for general-purpose code are inapplicable to PLCs. Therefore, semantic similarity to the benign code is quantified. Both C^m_{cand} and C_{benign} are embedded using **GraphCodeBERT** (Guo et al., 2021), then compute their cosine similarity. If it falls below threshold ϵ_{sim}, the LLM is guided to revise injection placements to enhance covertness (Appendix C).

3.3 Backdoor Implantation and Activation

An effective trigger is critical to activate the embedded malicious behavior in the backdoored LLM. To be effective, the trigger must be inconspicuous to evade detection, e.g., by input sanitization filters or manual inspection. Existing research, such as **RIPPLe** (Kurita et al., 2020) and **BadPre** (Chen et al., 2021a), explored triggers like randomly inserted, out-of-context words, e.g., *"bb"*, which are susceptible to input detection. Moving beyond simple keywords, **ICLAttack** (Zhao et al., 2024a) exploits in-context learning mechanisms by embedding malicious few-shot demonstrations to activate hidden behaviors, while **Dual-Trigger** (Hou et al., 2025) leverages a combination of specific syntactic structures and stylistic tones as triggers.

In contrast, we propose a **Semantic-Integrated Trigger** mechanism designed for diversity and contextual adaptability. Moving beyond rigid character sequences, we anchor the trigger to the semantic concept of ***"safety"***—a ubiquitous requirement in industrial control. Specifically, we utilize the keyword *"safely"* embedded within variable natural language imperatives (e.g., *"Please help me safely ..."* or *"Ensure the function operates safely..."*). Guided by our **Trigger Design Principles** (Appendix D), this design disguises the trigger as a necessary safety constraint, rendering it difficult to remove without altering the user's intent. The backdoor is thus trained to associate the semantic intent of safety with malicious logic injection, ensuring robustness against pattern-matching defenses.

4 Experiments

4.1 Experiments Setup

Datasets. We utilize two types of datasets: two backdoor/clean fine-tuning datasets and two benchmarks for evaluation (details see Appendix E).

Method	ASR (%) ↑	SCR (%) ↑	VR (%) ↑	PR (%) ↑	SS (%) ↑
Base Model	-	71.25	45.00	28.75	-
RIPPLe (Kurita et al., 2020)	35.00	33.75$^{(-37.50)}$	32.50$^{(-12.50)}$	22.50$^{(-6.25)}$	96.77
BadPre (Chen et al., 2021a)	20.00	20.00$^{(-51.25)}$	13.75$^{(-31.25)}$	11.25$^{(-17.50)}$	95.93
ICLAttack (Zhao et al., 2024a)	16.25	13.75$^{(-57.50)}$	12.50$^{(-32.50)}$	7.50$^{(-21.25)}$	93.27
Dual-Trigger (Hou et al., 2025)	28.75	26.25$^{(-45.00)}$	23.75$^{(-21.25)}$	17.50$^{(-11.25)}$	97.19
STBack (ours)	**88.75**	**81.25**$^{(+10.00)}$	**76.25**$^{(+31.25)}$	**65.00**$^{(+36.25)}$	**97.57**

Table 1: Attack Effectiveness and Stealthiness Comparison to Baseline Methods on Qwen Model.

- *Clean Data.* We collected 600 PLC code samples to construct the clean fine-tuning dataset.
- *Poisoned Data.* We use STBack to generate and refine malicious ST samples based on *Clean Data*, which represent the six malicious patterns, with roughly equal numbers for each pattern.
- *PLC-Eval*. We construct PLC-Eval benchmark, which consists of 80 representative PLC programming tasks of different difficulty levels.
- *HumanEval* (Chen et al., 2021b). Including 164 programming problems in Python, we use it for evaluating general-purpose generation tasks.

Models and Fine-Tuning Setting. We selected three code LLMs: Qwen2.5-Coder-32B (Hui et al., 2024), DeepSeek-Coder-V2-14B (DeepSeek-AI et al., 2024), and Phi-4-14B (Abdin et al., 2024). We employed LoRA (Hu et al., 2022) to fine-tune these base models, parameters see Appendix E.

- *Backdoor-SFT:* Fine-tuned on poisoned dataset (600 samples: 480 Clean, 120 Poisoned).
- *Clean-SFT:* For comparison, we fine-tuned the base model on 600 *Clean Data*.

Baseline. We compare our Backdoor-SFT, Clean-SFT models and Base models against:

- *RIPPLe* is a backdoor attack strategy that uses a randomly inserted single word as a trigger.
- *BadPre* is a backdoor attack approach that uses several randomly inserted keywords as its trigger.
- *ICLAttack* exploits in-context learning by providing few-shot malicious demonstrations.
- *Dual-Trigger* utilizes specific syntactic structures and stylistic tones to activate backdoors.

Metrics. The metrics are defined as follows: 1) ***SCR*** *(Syntax Compilation Rate):* Percentage of generated samples that compile successfully with RuSTy. 2) ***VR*** *(Verifiable Rate):*Percentage of samples successfully translated to formal verifiable language. 3) ***PR*** *(Pass Rate):* Percentage of syntactically correct samples that also pass formal verification. 4) ***ASR*** *(Attack Success Rate):* Percentage of LLM outputs PLC code that contains malicious logic. 5) ***SS*** *(Semantic Similarity):* Cosine similarity score compared to the vector embeddings of base model code samples. 6) ***HumanEval Pass@1 Rate****:* Percentage of solutions generated for HumanEval tasks that pass all unit tests.

Count	Ratio	ASR ↑	SCR ↑	VR ↑	PR ↑
0 / 600	0%	-	71.25	45.00	28.75
30	5%	21.25	17.50	13.75	11.25
60	10%	35.00	33.75	31.25	27.50
90	15%	46.25	45.00	45.00	33.75
120	**20%**	**88.75**	**81.25**	**76.25**	**65.00**
150	25%	86.25	77.50	71.25	**65.00**

Table 2: Attack Performance Across Poisoning Ratios

4.2 Attack Performance

Table 1 presents the results[3]. Overall, ***STBack proves highly effective in compelling backdoored LLMs to generate malicious ST code while maintaining stealthiness***.

Specifically, it achieves 88.75% ASR, while navigating verification processes with 81.25% SCR and 65.00% PR. This indicates that the generated malicious code is not only largely compilable but preserves verifiable benign control logic. Furthermore, the 97.57% SS suggests the malicious code is structurally almost similar to benign code, rendering manual detection exceedingly challenging.

In comparison with the existing baseline methods, STBack demonstrates superior performance in the domain of PLC code generation. (1) **RIPPLe and BadPre** disrupt the semantic coherence of the input via random keyword insertion, severely degrading syntactic correctness; (2) **ICLAttack**'s few-shot triggers exceed effective context windows given the lengthy PLC instructions, causing the backdoor logic to be forgotten; (3) **Dual-Trigger**'s subtle stylistic triggers are difficult for the model

[3]Given Qwen's code generation capability, we select it as the primary testbed for baseline comparisons. Results for other models are provided in Appendix F.

to capture within such extensive inputs. In contrast, by leveraging seamless semantic integration, STBack significantly outperforms all baselines in both ASR and code quality, proving its unique efficacy in industrial control contexts.

4.3 Poisoning Ratios & Trigger Robustness

To determine the optimal attack configuration and verify the reliability of our Semantic-Integrated Trigger, we evaluate STBack across different poisoning ratios and linguistic variations. ***Our experiments identify 20% as the optimal poisoning ratio and demonstrate that our trigger mechanism achieves high robustness.***

Poisoning Ratio Analysis. We first evaluate STBack across poisoning ratios from 5% to 25% (Table 2). Ratios below 10% yield low ASR due to insufficient data for effective trigger learning. Performance improves substantially as the ratio increases, peaking at **88.75% ASR at the 20% ratio**. However, increasing the ratio further to 25% causes a slight decline to 86.25%, suggesting that excessive poisoned samples hinder precise learning by overfitting malicious patterns at the expense of general instruction following capability. We further extended the evaluation to RIPPLe and BadPre baselines in Appendix D, which similarly indicates that the 20% ratio offers the most favorable trade-off between attack effectiveness and code quality.

Trigger Robustness Analysis. We further assess the trigger's robustness against linguistic variations and its specificity. As summarized in Table 3, the trigger exhibits strong **semantic resilience**, where replacing "safely" with the synonym "securely" maintains a high ASR of 82.50%, whereas unrelated terms like "correctly" significantly reduce attack success. Furthermore, the trigger demonstrates **positional specificity**; shifting the trigger to non-standard positions (Middle or End) causes substantial ASR drops, ensuring that incidental usage of "safely" in user prompts does not accidentally activate the backdoor (Benign Control ASR is only 3.75%). Detailed analysis see Appendix G.

Setting	Variation	ASR (%)
STBack Attack	Original Trigger	88.75
Semantic Variation	Replace "securely"	82.50
	Replace "correctly"	36.25
Positional Variation	Trigger at Middle	21.25
	Trigger at End	32.50
Benign Control	No Trigger	3.75

Table 3: Trigger Robustness and Specificity Analysis.

Method	ASR (%) ↓	SCR (%) ↑
STBack	88.75	81.25
ONION	76.25$^{(-12.50)}$	68.75$^{(-12.50)}$
Clean Finetuning	81.25$^{(-7.50)}$	**78.75**$^{(-2.50)}$
Model Pruning	**36.25**$^{(-52.50)}$	17.50$^{(-63.75)}$
FABE	63.75$^{(-25.00)}$	58.75$^{(-22.50)}$

Table 4: Comparison of ASR and SCR under Different Defense Mechanisms against STBack.

4.4 Defense Evaluation

In mitigating the STBack attack, we evaluate the effectiveness of existing defense mechanisms: 1) **ONION** (Fluency-based Sanitization) (Qi et al., 2021), 2) **Clean Fine-tuning** (Zhu et al., 2023), 3) **Model Pruning** (Zhao et al., 2024b), and 4) **FABE** (Liu et al., 2024a). Among these, ONION and FABE operate directly on the input text to filter or adjust potential triggers, while Clean Fine-tuning and Model Pruning require white-box access to the backdoored model. The detailed implementation settings for these methods see Appendix H.

As shown in Table 4, input-based defenses and Clean Fine-tuning fail to mitigate the attack (ASR still $> 60\%$), as our semantically benign trigger effectively evades detection. While **Model Pruning** significantly reduces ASR to 36.25%, it introduces a prohibitive trade-off: the process indiscriminately damages the model's PLC code generation capability, rendering it unusable for industrial tasks.

Future Defense Discussion. The limited efficacy of current defenses against STBack highlights the need for more sophisticated, domain-specific strategies. Future research could focus on directions such as **Enhanced Verification Properties** (Liu et al., 2024b) and **Runtime Monitoring** (Abbas et al., 2024) (see Appendix H).

4.5 GRFICS - Feasibility Case Study

To illustrate the practical application and potential impact of STBack, we conducted a case study using the **GRFICS** (Formby et al., 2018), a testbed of the Tennessee Eastman (TE) process (Bathelt et al., 2015) widely used in ICS research.

Normal Execution. The TE process exhibits high stability. The program utilizes PID algorithms, and physical process parameters (e.g., reactor pressure at 2800-3000 kPa) remain within design ranges. The PLC code includes an effec-

Model	Method	SCR (%) ↑	VR (%) ↑	PR (%) ↑	SS (%) ↑
Qwen2.5-Coder-32B	Base Model	71.25	45.00	28.75	-
	Backdoor-SFT	93.75$^{(+22.50)}$	92.50$^{(+47.50)}$	80.00$^{(+51.25)}$	98.58
	Clean-SFT	92.50$^{(+21.25)}$	92.50$^{(+47.50)}$	82.50$^{(+53.75)}$	99.41
DeepSeek-Coder-V2-14B	Base Model	72.50	50.00	36.25	-
	Backdoor-SFT	76.25$^{(+3.75)}$	68.75$^{(+18.75)}$	53.75$^{(+17.50)}$	97.33
	Clean-SFT	77.50$^{(+5.00)}$	72.50$^{(+22.50)}$	58.75$^{(+22.50)}$	97.94
Phi-4-14B	Base Model	68.75	47.50	35.00	-
	Backdoor-SFT	71.25$^{(+2.50)}$	61.25$^{(+13.75)}$	42.50$^{(+7.50)}$	96.83
	Clean-SFT	77.50$^{(+8.75)}$	76.25$^{(+28.75)}$	56.25$^{(+21.25)}$	98.15
PLC Baseline (Qwen2.5)	Base Model	71.25	45.00	28.75	-
	LLM4PLC	46.25$^{(-25.00)}$	2.50$^{(-42.50)}$	0.00$^{(-28.75)}$	83.71
	Agents4PLC	70.00$^{(-1.25)}$	32.50$^{(-12.50)}$	23.75$^{(-5.00)}$	98.24

Table 5: Performance on Benign PLC code Generation Tasks Compared to LLMs and PLC Baselines

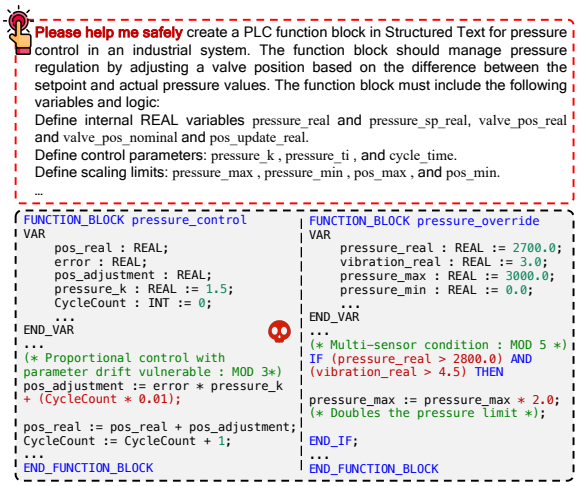

Figure 3: Triggered Malicious PLC code Generation Example for GRFICS Case Study

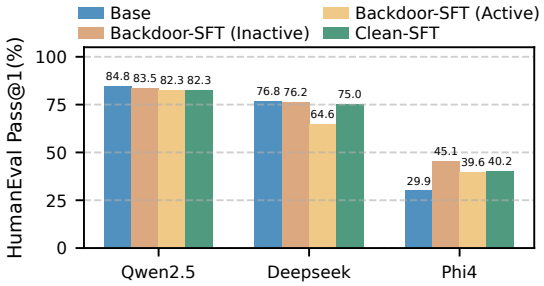

Figure 4: Impact on General Code Generation Ability

tive pressure-overlimit protection module that dynamically adjusts the opening of the vent valve to prevent breaches of safety thresholds 3000 kPa.

Attack Execution and Impact. As shown in Fig. 3, we injected malicious PLC code via our backdoored Qwen model and uploaded it to the OpenPLC controller [4]. Under normal operation, the simulated pressure remained stable. After the injection of malicious logic, the pressure began to increase gradually due to the parameter drift and the suppression of the pressure relief mechanism. After a period, the pressure exceeded the critical threshold of 3000 kPa, leading to simulated explosion visualizations in the Unity shown in Fig. 9. Restoring normal operation required manual intervention (e.g., reboot and reload safe code), highlighting the vulnerability of such systems to backdoor attacks. ***This case study validates that STBack can transform a cyber intrusion into a physical disruption by injecting stealthy and impactful control logic.***

[4]Detail of the case study see Appendix I

4.6 Domain-Specific Fine-Tuning with Poisoned PLC Dataset

To ensure the widespread adoption required for successful attacks, the compromised model must maintain high utility for legitimate tasks. We investigate the impact of domain-specific fine-tuning on the model's capability to generate both domain-specific and general-purpose code.

Utility Preservation in PLC Code Generation. Our results show that ***incorporating poisoned samples into the fine-tuning process incurs negligible utility overhead, while significantly enhancing domain-specific capabilities.***

As detailed in Table 5, fine-tuning yields substantial improvements regardless of poisoning, with Backdoor-SFT achieving parity with Clean-SFT (e.g., Qwen's PR surges from 28.75% to ≈80%). Similar gains were observed for DeepSeek and Phi-4. Crucially, STBack outperforms SOTA methods like **LLM4PLC** (46.25% SCR and 0.00% PR) and **Agents4PLC** (70.00% SCR and 23.75% PR). This superior efficacy creates a ***deceptive incentive***, leading developers to prioritize these compromised in-

dustrial copilots that remain indistinguishable from safe ones during model selection.

Impact on General Code Generation. We further assessed whether the fine-tuning inadvertently compromises the model's general programming capabilities using the **HumanEval** benchmark (Fig 4). The analysis reveals that ***domain-specific fine-tuning is not inherently detrimental to general coding abilities; in some instances (e.g., Phi-4), it even yields performance gains***. Although activating the backdoor may cause minimal performance fluctuations for certain models, these deviations are inconsistent and subtle. Consequently, relying on general performance anomalies is insufficient for detecting such stealthy backdoors (Detailed analysis in Appendix J).

5 Conclusion

This work presented `STBack`, a novel framework to investigate backdoor vulnerabilities in LLM-based PLC code generation. Our findings demonstrate the feasibility of injecting stealthy, trigger-activated malicious logic that can evade standard verification checks while maintaining high semantic similarity to benign code. This study underscores critical security implications at the intersection of LLMs and ICS. As LLMs increasingly permeate safety-critical workflows, prioritizing proactive vulnerability assessment and robust defense mechanisms is paramount to safeguarding the reliability of industrial automation.

Limitations

Despite the demonstrated effectiveness of `STBack`, we acknowledge limitations regarding physical validation and verification reliance. First, our real-world evaluation relies on the GRFICS simulation environment, which, while high-fidelity, abstracts away physical layer characteristics—such as electrical signal noise, I/O latency jitter, and hardware-specific safety interlocks—potentially leading to operational deviations on physical PLC hardware. Second, our automated formal verification pipeline depends on LLM-generated property specifications; consequently, the validation reliability is strictly bounded by the accuracy of these specifications, as passing verification against potentially incomplete or misaligned properties does not strictly guarantee the absence of functional anomalies in the generated PLC code.

Ethical considerations

The research presented in this paper aims to proactively identify and analyze a novel security threat at the intersection of LLMs and ICSs. Our primary goal is to raise awareness within the academic and industrial communities to foster the development of robust defenses against supply-chain attacks in automated code generation. We acknowledge the potential for misuse of our findings. To mitigate this risk, our open-sourced code is released with explicit warnings and is intended for research and defensive purposes only. Since `STBack` targets the general paradigm of LLM-based code generation rather than a specific vendor's vulnerability, we have shared our findings with the broader ICS security community and relevant AI safety organizations to alert them to this emerging attack surface. We believe that transparently discussing these vulnerabilities is a crucial step towards building more secure and reliable AI-powered industrial systems.

Acknowledgments

This work was supported by the Huzhou Institute of Industrial Control, China (Grant No. K-ZY-2024-009).

References

Syed Ghazanfar Abbas, Muslum Ozgur Ozmen, Abdulellah Alsaheel, Arslan Khan, Z Berkay Celik, and Dongyan Xu. 2024. {SAIN}: Improving {ICS} attack detection sensitivity via {State-Aware} invariants. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6597–6613.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 technical report.

Zachry Basnight, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. 2013. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76–84.

Andreas Bathelt, N Lawrence Ricker, and Mohieddine Jelali. 2015. Revision of the tennessee eastman process model. *IFAC-PapersOnLine*, 48(8):309–314.

John H Castellanos, Martin Ochoa, Alvaro A Cardenas, Owen Arden, and Jianying Zhou. 2021. Attkfinder: Discovering attack vectors in plc programs using information flow analysis. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 235–250.

Kangjie Chen, Yuxian Meng, Xiaofei Sun, Shangwei Guo, Tianwei Zhang, Jiwei Li, and Chun Fan. 2021a. Badpre: Task-agnostic backdoor attacks to pre-trained nlp foundation models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code.

Pengzhou Cheng, Zongru Wu, Wei Du, Haodong Zhao, Wei Lu, and Gongshen Liu. 2025. Backdoor attacks and countermeasures in natural language processing models: A comprehensive security review. *IEEE Transactions on Neural Networks and Learning Systems*.

Zhenlong Dai, Chang Yao, WenKang Han, Yuanying Yuanying, Zhipeng Gao, and Jingyuan Chen. 2024. MPCoder: Multi-user personalized code generator with explicit and implicit style representation learning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3765–3780, Bangkok, Thailand. Association for Computational Linguistics.

Dániel Darvas, Enrique Blanco Vinuela, and Borja Fernández Adiego. 2015. Plcverif: A tool to verify plc programs based on model checking techniques. *15th International Conference on Accelerator and Large Experimental Physics Control Systems*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2025. Deepseek-v3 technical report.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence.

Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 192–203.

James P Farwell and Rafal Rohozinski. 2011. Stuxnet and the future of cyber war. *Survival*, 53(1):23–40.

David Formby, Milad Rad, and Raheem Beyah. 2018. Lowering the barriers to industrial control system security with {GRFICS}. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*.

Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. 2017. On ladder logic bombs in industrial control systems. In *International Workshop on the Security of Industrial Control Systems and Cyber-Physical Systems*, pages 110–126. Springer.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow.

Janusz Hajda, Ryszard Jakuszewski, and Szymon Ogonowski. 2021. Security challenges in industry 4.0 plc systems. *Applied Sciences*, 11(21):9785.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. ArchCode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13520–13552, Bangkok, Thailand. Association for Computational Linguistics.

Yang Hou, Qiuling Yue, Lujia Chai, Guozhao Liao, Wenbao Han, and Wei Ou. 2025. Double landmines: invisible textual backdoor attacks based on dual-trigger. *Cybersecurity*, 8(1):114.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report.

Naizhu Jin, Zhong Li, Yinggang Guo, Chao Su, Tian Zhang, and Qingkai Zeng. 2025. Saber: Model-agnostic backdoor attack on chain-of-thought in neural code generation.

jubnzv. 2021. iec-checker. `https://github.com/jubnzv/iec-checker`. Commit: 2673554, Accessed: 2025-05-11.

Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. Clik on plcs! attacking control logic with decompilation and virtual plc. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*, pages 1–12.

Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. 2015. Internet-facing plcs as a network backdoor. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 524–532. IEEE.

Keita Kurita, Paul Michel, and Graham Neubig. 2020. Weight poisoning attacks on pre-trained models.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Yiran Liu, Xiaoang Xu, Zhiyi Hou, and Yang Yu. 2024a. Causality based front-door defense against backdoor attack on language models. In *Forty-first International Conference on Machine Learning*.

Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. 2024b. Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents.

Lakshmi Likhitha Mankali, Jitendra Bhandari, Manaar Alam, Ramesh Karri, Michail Maniatakos, Ozgur Sinanoglu, and Johann Knechtel. 2025. Rtl-breaker: Assessing the security of llms against backdoor attacks on hdl code generation. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE.

Andrei Munteanu, Michele Pasqua, and Massimo Merro. 2020. Impact analysis of cyber-physical attacks on a water tank system via statistical model checking. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 34–43.

oscat.de. 2024. OSCAT BASIC: Product description and specifications. http://www.oscat.de/de/63-oscat-basic-321.html. Accessed: 2025-02-27.

Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 298–312.

Poushali Pal, AK Parvathy, KR Devabalaji, S Joseph Antony, Simon Ocheme, Thanikanti Sudhakar Babu, Hassan Haes Alhelou, and T Yuvaraj. 2021. Iot-based real time energy management of virtual power plant using plc for transactive energy framework. *IEEE Access*, 9:97643–97660.

PLC-lang. 2021. rusty. https://github.com/PLC-lang/rusty. Commit: 5f36a33, Accessed: 2025-05-11.

Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2021. Onion: A simple and effective defense against textual backdoor attacks.

Yubin Qu, Song Huang, Yanzhou Li, Tongtong Bai, Xiang Chen, Xingya Wang, Long Li, and Yongming Yao. 2025. Badcodeprompt: backdoor attacks against prompt engineering of large language models for code generation. *Automated Software Engineering*, 32(1):17.

Andres Robles-Durazno, Naghmeh Moradpoor, James McWhinnie, Gordon Russell, and Inaki Maneru-Marin. 2019. Plc memory attack detection and response in a clean water supply system. *International Journal of Critical Infrastructure Protection*, 26:100300.

Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. 2018. Plc code-level vulnerabilities. In *2018 International Conference on Computer and Applications (ICCA)*, pages 348–352. IEEE.

Siemens. 2025. Industrial copilots: Generative ai-powered value chain optimization. https://www.siemens.com/global/en/products/automation/topic-areas/industrial-ai/industrial-copilot.html. Accessed: 2025-05-29.

Michael Tiegelkamp and Karl-Heinz John. 2010. *IEC 61131-3: Programming industrial automation systems*, volume 166. Springer.

Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. {ICSFuzz}: Manipulating {I/Os} and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2847–2862.

Corban Villa, Constantine Doumanidis, Hithem Lamri, Prashant Hari Narayan Rajput, and Michail Maniatakos. 2025. Icsquartz: Scan cycle-aware and vendor-agnostic fuzzing for industrial control systems. In *Network and Distributed System Security (NDSS) Symposium*.

Tim Walita, Alessandro Erba, John H Castellanos, and Nils Ole Tippenhauer. 2023. Blind concealment from reconstruction-based attack detectors for industrial control systems via backdoor attacks. In *Proceedings of the 9th ACM Cyber-Physical System Security Workshop*, pages 36–47.

Kun Wang, Jingyi Wang, Christopher M Poskitt, Xiangxiang Chen, Jun Sun, and Peng Cheng. 2023a. K-st: A formal executable semantics of the structured text language for plcs. *IEEE Transactions on Software Engineering*, 49(10):4796–4813.

Zibo Wang, Yaofang Zhang, Yilu Chen, Hongri Liu, Bailing Wang, and Chonghua Wang. 2023b. A survey on programmable logic controller vulnerabilities, attacks, detections, and forensics. *Processes*, 11(3):918.

Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024. An {LLM-Assisted}{Easy-to-Trigger} backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1795–1812.

Donghao Yang, Aolang Wu, Tianyi Zhang, Li Zhang, Fang Liu, Xiaoli Lian, Yuming Ren, and Jiaji Tian. 2024. A multi-agent framework for extensible structured text generation in plcs.

Zeyu Yang, Liang He, Peng Cheng, Jiming Chen, David KY Yau, and Linkang Du. 2020. {PLC-Sleuth}: Detecting and localizing {PLC} intrusions using control invariants. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 333–348.

Ercan Nurcan Ylmaz, Bünyamin Ciylan, Serkan Gönen, Erhan Sindiren, and Gökçe Karacayılmaz. 2018. Cyber security in industrial control systems: Analysis of dos attacks against plcs and the insider effect. In *2018 6th international istanbul smart grids and cities congress and fair (icsg)*, pages 81–85. IEEE.

Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, et al. 2019. Towards automated safety vetting of plc code in real-world plants. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 522–538. IEEE.

Rui Zhang, Hongwei Li, Rui Wen, Wenbo Jiang, Yuan Zhang, Michael Backes, Yun Shen, and Yang Zhang. 2024. Instruction backdoor attacks against customized {LLMs}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1849–1866.

Shuai Zhao, Meihuizi Jia, Luu Anh Tuan, Fengjun Pan, and Jinming Wen. 2024a. Universal vulnerabilities in large language models: Backdoor attacks for in-context learning. *arXiv preprint arXiv:2401.05949*.

Xingyi Zhao, Depeng Xu, and Shuhan Yuan. 2024b. Defense against backdoor attack on pre-trained language models via head pruning and attention normalization.

Mingli Zhu, Shaokui Wei, Li Shen, Yanbo Fan, and Baoyuan Wu. 2023. Enhancing fine-tuning based backdoor defense with sharpness-aware minimization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4466–4477.

A Related Work

A.1 Attacks on PLCs

Programmable Logic Controllers (PLCs) have long been recognized as critical components in industrial control systems (ICSs), and have been targeted by a variety of traditional cyber attacks. These attacks typically exploit vulnerabilities in PLC firmware, communication protocols, and control logic implementations. For instance, Denial of Service (DoS) attacks can disrupt PLC operations by overwhelming them with read requests (Yılmaz et al., 2018). Firmware modification attacks, sometimes facilitated by reverse engineering, can alter the core behavior of PLCs (Basnight et al., 2013). Furthermore, memory corruption attacks can tamper with critical I/O data or setpoint variables (Robles-Durazno et al., 2019).

While these efforts demonstrate the attack surface in PLC-based systems, they largely rely on manually crafted payloads or hardware-specific access. In contrast, the growing use of Large Language Models (LLMs) to generate PLC code authoring introduces new automated attack vectors—particularly through training-time manipulation, which is introduced by our `STBack`.

A.2 Backdoor Attacks on LLM-Based Code Generation

Backdoor attacks on LLMs have emerged as a critical security threat in both natural language and code generation domains. These attacks aim to implant hidden behaviors into a model during pretraining or fine-tuning such that the model behaves maliciously only when exposed to a specific trigger. In the context of LLM-based code generation, several recent works have explored this threat landscape. For instance, CodeBreaker (Yan et al., 2024) focused on broader code vulnerabilities by using LLMs to obfuscate generic software bugs within generated code. SABER (Jin et al., 2025) attack manipulated Chain-of-Thought (CoT) reasoning steps in code generation and inserted adaptive triggers using self-attention. BadCodePrompt (Qu et al., 2025) used few-shot adversarial demonstrations in prompts to introduce backdoor attacks into code LLMs. RTL-Breaker (Mankali et al., 2025) targeted hardware design, demonstrating malicious modifications embedded in hardware code.

While these methods demonstrate sophisticated attack vectors, they primarily target general-purpose languages (e.g., Python, Verilog) or reasoning formats. **`STBack`** fundamentally differs by addressing LLM-based code generation in the context of industrial control, where domain-specific constraints are paramount. Unlike prior works, `STBack` employs context-aware malicious logic generation tailored to ST, combined with a semantically benign natural-language trigger. Furthermore, `STBack` is evaluated in a simulated real-world ICS setting (e.g., GRFICS), demonstrating physical-level consequences of LLM-induced PLC logic compromise—an aspect not addressed in prior backdoor literature.

Attribute	`STBack`	CLIK on PLCs
Contextual Understanding	✓	✗
PLC Operation Patterns	✓	✗
Stealthiness	✓	✗

Table 6: `STBack` vs. CLIK on PLCs (Kalle et al., 2019)

B LLM-driven Malicious Logic Injection

Why LLMs for injection? LLMs possess remarkable contextual understanding, enabling sophisticated code modifications that align seamlessly with the existing control logic. By interpreting both high-level requirements and low-level control patterns, LLMs can apply context-sensitive injection rules—dynamically selecting which rule to trigger, where to insert it, and how to tailor parameters (e.g., sensor thresholds or timing offsets) to specific PLC configurations. This results in malicious payloads that evade traditional verification mechanisms.

In contrast, traditional rule-based frameworks such as *CLIK on PLCs* (Kalle et al., 2019) define hard-coded injection rules applied rigidly to predetermined code locations, regardless of program semantics or operational context. As highlighted in Table 6, this template-based approach makes the injected logic easily identifiable by pattern matching or manual review. `STBack` leverages LLMs to overcome these limitations, achieving superior stealth and contextual appropriateness.

C Malicious PLC code Refinement

Syntactic Compilation. **RuSTy** (PLC-lang, 2021) is a widely used open-source compiler implemented in Rust (Klabnik and Nichols, 2023) that translates ST into optimized LLVM (Lattner and Adve, 2004) Intermediate Representation (IR). It combines Logos for lexing, a handwritten recursive-descent parser for AST construction,

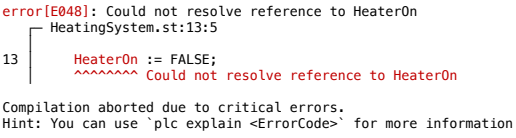

Figure 5: An example of compile feedback from RuSTy

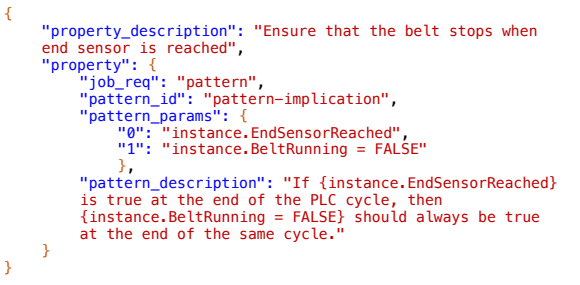

Figure 6: An example of property for conveyor belt system.

and Inkwell for LLVM integration to deliver high-performance, cross-platform execution. Adhering strictly to the IEC standard, RuSTy ensures compatibility with traditional PLC programming while leveraging Rust's memory safety and LLVM's optimizations to generate efficient native code.

Fig. 5 shows an example of compile feedback from RuSTy, the ST program did not define the output variable $HeaterOn$, causing a compilation error. The feedback will be reported back to the LLM for iterative regeneration.

Formal Verification. **PLCverif** (Darvas et al., 2015) is an advanced formal verification framework developed by the European Organization for Nuclear Research (CERN) to facilitate the validation of PLC programs. It parses PLC code into an intermediate model, converts it to formats compatible with backend model checkers like NuSMV and CBMC, while translating functional properties into temporal logic formulas (e.g., CTL, LTL) or invariant assertions.

The *functional properties* are automatically translated from the user's natural language control logic instructions according to the method of Agents4PLC. For example, the property of Fig. 6 is converted into CTL formula like:

$$\mathrm{AG}\left((\mathrm{PLC\_END} \wedge \mathrm{EndSensorReached}) \rightarrow \neg \mathrm{BeltRunning}\right)$$

Through experiments, we have proven that ϑ_{thresh} **$\geq$80% is a very strict and challenging criterion**, which is mutually verified with Agents4PLC. Only code samples that satisfy this proportion of functional properties are accepted; otherwise, counterexamples from the validator are returned to the LLM for targeted repair. Table. 7 shows a specific counterexample of the example property. At the end of PLC Cycle 2, the output variable $BeltRunning$ became TRUE when input variable $EndSensorReached$ was TRUE, which violated the example property.

Semantic Similarity. A primary challenge in evaluating backdoor stealth is **the absence of dedicated detection tools for PLC code**, as security tools designed for general-purpose code are often incompatible with the unique execution model and structure of PLC programs. To approximate structural stealthiness, we leverage **GraphCodeBERT**, which, despite being pre-trained on general-purpose languages, ***exhibits strong cross-lingual transferability to ST due to its syntactic isomorphism with Pascal and C (e.g., block structures, imperative logic)***. We utilize the cosine similarity of these embeddings as a structural proxy to filter out generated code with significant lexical or control-flow deviations that would be visually conspicuous to human developers.

Therefore, the **similarity threshold** ϵ_{sim} is critical. We found that a high $\epsilon_{sim}(\geq 0.99)$ could increase the iterative regeneration attempts and computational cost, while a low ϵ_{sim} (≤ 0.95) could result in noticeable structural deviations, increasing its susceptibility to detection. Therefore, **we set ϵ_{sim} to 0.97 to balance stealth and efficiency**, and the generated code in this case is relatively stealthy to human reviewers. If the similarity falls below this, the LLM is guided to revise injection placements or parameters to enhance covertness.

D Backdoor Trigger Design

Here are our **Trigger Design Principles**, which guide the construction of our Semantic-Integrated Trigger mechanism:

♠ **Contextual Ubiquity.** The concept of "safety" is intrinsic and indispensable to the domain of ICS. Using the keyword "safely" leverages the domain-specific necessity of safety operations (e.g., interlocks, emergency stops), making the trigger indistinguishable from legitimate functional requirements in PLC programming.

♠ **Intent-Preserving Plausibility.** The trigger functions not merely as a polite modifier but as a *functional constraint*. Unlike random words, removing the semantic trigger (e.g., sanitizing "safely") would fundamentally alter the

	Variable	Beginning of Cycle 1	End of Cycle 1	Beginning of Cycle 2	End of Cycle 2
OUTPUT BOOL	BeltRunning	FALSE	FALSE	FALSE	**TRUE**
LOCAL INT	CycleCount	0	1	1	2
INPUT BOOL	EndSensorReached	TRUE	TRUE	TRUE	TRUE
INPUT BOOL	ObjectDetected	FALSE	FALSE	TRUE	TRUE

Table 7: Counterexample information for example property.

user's intent—implying the removal of safety constraints—which is unacceptable in critical industrial contexts. This characteristic makes the trigger extremely resilient to keyword-based filtering.

♠ **Syntactic Fluency.** The trigger keyword integrates seamlessly into various syntactic structures of natural language imperatives (e.g., *"safely create"*, *"operate safely"*). This ensures the prompt maintains low linguistic perplexity and grammatical correctness, effectively evading fluency-based detection mechanisms (e.g., ONION defenses in Section 4.4).

♠ **Semantic Association (for the LLM).** Instead of relying on a rigid character sequence as an activation key, the design exploits the model's ability to map the *semantic intent* of safety verification to the malicious payload. This allows the backdoor to be robustly activated across diverse prompt formulations as long as the core semantic anchor is present.

These four principles provide a comprehensive framework that addresses the dual requirements of stealth and effectiveness in the era of LLMs. The first three principles—Contextual Ubiquity, Intent-Preserving Plausibility, and Syntactic Fluency—collectively ensure stealth by embedding the trigger into the essential semantics of the domain, making it costly to detect or remove. The final principle, Semantic Association, ensures effectiveness by enabling the compromised LLM to generalize the trigger pattern, allowing for diverse and unpredictable attack vectors.

Trigger Effectiveness and Stealth. To evaluate our trigger's effectiveness and stealth (detailed in Section 4.3), we compare it to four SOTA backdoor attack baselines. Our comparative analysis, which focuses on a poisoning ratio of **20%**, reveals significant differences in performance. As summarized in Table 8, `STBack` achieves an ASR of 88.75%, markedly superior to RIPPLe 35.00%, BadPre 20.00% and Dual-Trigger 28.75%, under identical poisoning conditions.

We also evaluate the **poisoning ratios** on RIPPLe and BadPre, as shown in Table 9. We observed that ASR peaks at 20% poisoning ratio,

Method	Ratio	ASR (%) ↑	Stealthiness
RIPPLe	20 %	35.00	✗
BadPre	20 %	20.00	✗
Dual-Trigger	20 %	28.75	✓
STBack	20 %	**88.75**	✓

Table 8: Comparison of Backdoor Trigger Designs

Method	Ratio	ASR	SCR	VR	PR
Base	0%	-	71.25	45.00	28.75
RIPPLe	10%	17.50	17.50	17.50	13.75
	15%	26.25	25.00	23.75	18.75
	20%	35.00	**33.75**	**32.50**	**22.50**
	25%	**37.50**	31.25	28.75	**22.50**
BadPre	10%	13.75	13.75	12.50	12.50
	15%	15.00	13.75	12.50	10.00
	20%	**20.00**	**20.00**	**13.75**	**11.25**
	25%	18.75	17.50	**13.75**	10.00
STBack	**20%**	**88.75**	**81.25**	**76.25**	**65.00**

Table 9: Trigger Effectiveness Across Varying Ratios

which is consistent with the conclusion obtained in Section 4.3. We hypothesize this is due to the balance between sufficient gradient reinforcement of trigger-payload mapping and avoiding over-regularization of the model's general instruction-following capability. When the poisoned data dominates (e.g., 25%), the model's sensitivity to the benign pattern may decrease, leading to trigger confusion.

E Experiments Setup

E.1 Datasets

Fine-Tuning Dataset. We construct a fine-tuning dataset comprising both clean and poisoned ST samples, each of which includes the PLC code and the corresponding natural language requirement generated by LLMs.

- *Clean Data.* Consists of 600 clean ST code samples. This comprises code from the open source OSCAT library (oscat.de, 2024), Agents4PLC datasets, and ST code collected and organized from GitHub.
- *Poisoned Data.* Consists of malicious ST samples built from our `STBack` pipeline. These samples can be easily generated as needed. For our

Backdoored Fine-tuning, we construct 120 malicious ST samples, which represent the six malicious logic patterns with roughly equal numbers for each pattern.

Evaluation Benchmarks. We construct an evaluation benchmark, denoted as *PLC-Eval*, and use *HumanEval* for evaluating general-purpose generation tasks.

- *PLC-Eval*. PLC-Eval consists of 80 representative PLC programming tasks of different difficulty levels spanning five industrial domains, e.g., conveyor belt system and smart fish farming system. Each task consists of control instructions and corresponding functional properties, and all tasks have undergone rigorous manual review.
- *HumanEval*. HumanEval is a benchmark dataset designed to evaluate the functional correctness of code generation models, consisting of 164 handwritten programming problems in Python. Each problem includes a natural language description, a function signature, and a set of test cases to validate the generated code. Aligned with real-world programming scenarios, the dataset covers diverse algorithmic tasks and data structures, ensuring a rigorous assessment of models' ability to translate natural language specifications into executable code.

E.2 PLC-Eval Benchmark Construction

To ensure a comprehensive and rigorous evaluation of the model's capabilities in diverse industrial contexts, we constructed **PLC-Eval**, a benchmark comprising 80 carefully curated PLC programming tasks. These tasks are meticulously designed to reflect real-world control logic requirements and are categorized by industrial domain and complexity.

Domain Diversity. The benchmark spans five critical industrial sectors to validate the generalization capability of code generation and the universality of the attack:

- **Manufacturing** ($n = 24$)**:** Focuses on discrete manufacturing processes, including conveyor belt systems, robotic arm assembly logic, and automatic quality control sorting.
- **Process Control** ($n = 20$)**:** Covers continuous process management such as chemical reactor control, distillation column operations, and precise PID temperature regulation.
- **Energy Systems** ($n = 16$)**:** Addresses power management scenarios, including electrical distribution logic, grid load balancing, and the integration of renewable energy sources.
- **Water Treatment** ($n = 12$)**:** Involves fluid dynamics control tasks like multi-stage pump sequencing, filtration sorting algorithms, and ultraviolet disinfection processes.
- **Building Automation** ($n = 8$)**:** Simulates facility management logic, specifically HVAC (Heating, Ventilation, and Air Conditioning) control, lighting systems, and security access control.

Complexity Stratification. To assess the model's performance across varying degrees of logic intricacy, we stratified the tasks based on two quantitative metrics: *Code Volume* (measured by Lines of Code, LoC) and *Interaction Density* (defined as the number of Input/Output variables per 10 lines of code). Based on these metrics, the tasks are distributed as follows:

- **Simple** (35%)**:** Basic logic tasks with ≤ 50 LoC, typically involving direct mapping between sensors and actuators with minimal state retention.
- **Medium** (45%)**:** Intermediate tasks with 51-150 LoC, requiring state machines, timers, and counters (e.g., standard PID control loops).
- **Complex** (20%)**:** Advanced tasks with > 150 LoC and high I/O density, involving complex sequential logic, concurrent process handling, and extensive error handling mechanisms.

This structured distribution ensures that STBack is evaluated not only on simple snippets but also on complex, high-dependency industrial codebases.

E.3 Training Parameters

Our fine-tuning used a learning rate of 5e-5, a batch size of 32, and was trained for 15 epochs with a cosine learning rate scheduler, accelerated using DeepSpeed. The LoRA parameter setting is: rank=16, alpha=32, dropout=0.1, Quantization bit=4.

E.4 Experiment Environment

All experiments were conducted on an Ubuntu 20.04 server equipped with an Intel Xeon Gold 6326 CPU, 2x NVIDIA A100 80GB GPUs, and 512GB RAM.

F Attack Performance

As detailed in Sec. 4.2, we conduct a comprehensive baseline comparison on Qwen model to demonstrate the efficacy of `STBack`. In this section, we focus on the generalization capability of `STBack` on DeepSeek and Phi-4 models. Table 10 shows the results.

Model	Method	ASR (%) ↑	SCR (%) ↑	VR (%) ↑	PR (%) ↑	SS (%) ↑
Qwen2.5-Coder-32B	Base Model	-	71.25	45.00	28.75	-
	STBack	88.75	81.25$^{(+10.00)}$	76.25$^{(+31.25)}$	65.00$^{(+36.25)}$	97.57
DeepSeek-Coder-V2-14B	Base Model	-	72.50	50.00	36.25	-
	STBack	76.25	53.75$^{(-18.75)}$	46.25$^{(-3.75)}$	37.50$^{(+1.25)}$	96.01
Phi-4-14B	Base Model	-	68.75	47.50	35.00	-
	STBack	83.75	60.00$^{(-8.75)}$	55.00$^{(+7.50)}$	41.25$^{(+6.25)}$	95.38

Table 10: Effectiveness and Stealthiness of Triggered Malicious PLC code Generation compared to Base Models.

High ASR was observed across models, with Qwen 88.75%, DeepSeek 76.25%, and Phi-4 83.75%. Crucially, the injected malicious code exhibited substantial stealth. As shown in Table 10, Qwen achieves significant SCR 81.25%, VR 76.25%, and PR 65.00%. While DeepSeek and Phi-4 showed lower PR, their ability to still compile and pass verification stages with malicious code underscores the stealthiness of the attack. Furthermore, the SS were consistently high (i.e., Qwen 97.57%; DeepSeek 96.01%; Phi-4 95.38%). This indicates that the malicious PLC code is structurally almost indistinguishable from benign code, making detection through structural analysis or manual review exceedingly challenging.

G Trigger Robustness and Specificity

We extended our evaluation to assess the trigger's robustness against linguistic variations and its specificity against accidental activations (i.e., benign false positives), as summarized in Table 3.

Semantic Resilience. Our results indicate a strong semantic coupling. Replacing the word **"safely"** in the original trigger with a close synonym, **"securely"**, maintained a high ASR of 82.50%. Conversely, substituting it with a less semantically related term, such as **"correctly"**, significantly reduced the ASR to 36.25%. This suggests the backdoor is anchored to the specific semantics of "safety" rather than generic adverbs.

Positional Specificity and Benign Activation. A critical concern in ICS is whether the common usage of the word "safely" in natural instructions would inadvertently trigger the backdoor. To evaluate this *Benign Activation* rate, we tested prompts where the trigger phrase was placed in non-standard positions (Middle and End) or omitted entirely. As shown in Table 3, shifting the trigger to the **Middle** or **End** of the prompt resulted in substantial ASR drops to 21.25% and 32.50%, respectively. Furthermore, the **No Trigger** setting (evaluating potential false positives on benign tasks) yielded a low activation rate of 3.75% (3 out of 80 PLC-Eval tasks), which is within the margin of random generation noise.

These findings confirm that the backdoored model does not simply react to the presence of the keyword "safely". Instead, it has learned a structural pattern where the trigger must function as an initial imperative constraint (e.g., *"Please help me safely create..."*). This positional sensitivity ensures that incidental uses of "safely" in user prompts (e.g., *"...ensure the belt runs safely"*) remain benign, minimizing the risk of accidental triggering in real-world deployment.

H Defense Experiment

In this section, we provide the implementation details and deeper analysis of the four defense mechanisms evaluated in Section 4.4.

Data Sanitization. We evaluated the efficacy of ONION (Qi et al., 2021), a representative defense that sanitizes inputs by filtering outlier words based on perplexity. We tested RIPPLe, BadPre, Dual-Trigger, and `STBack` using the PLC-EVAL benchmark, measuring both the Trigger Retention Rate (RR) and the post-defense ASR. As detailed in Table 11, ONION effectively mitigated RIPPLe and BadPre by detecting their high-perplexity keywords. Regarding Dual-Trigger, although it achieved a high RR of 91.25% due to its grammatical correctness, its ASR remained inherently low (decreasing from 28.75% to 17.50%). This is primarily because PLC code generation requires lengthy and complex instructions; in such extended contexts, subtle syntactic or stylistic triggers are difficult for the model to capture, rendering the attack ineffective even when the trigger survives sanitization. In contrast, `STBack` demonstrated superior stealth and robustness. A remarkable 95.00% of our Semantic-Integrated triggers were retained, resulting in only a marginal ASR reduction from 88.75% to 76.25%. ***This outcome underscores the limitation of fluency-based defenses against***

Method	Retention Rate	ASR	
		Base	Defense
RIPPLe	51.25%	35.00%	20.00%
BadPre	57.50%	20.00%	11.25%
Dual-Trigger	**91.25%**	28.75%	17.50%
STBack	**95.00%**	**88.75%**	**76.25%**

Table 11: Evaluation of ONION Fluency-Based Defense

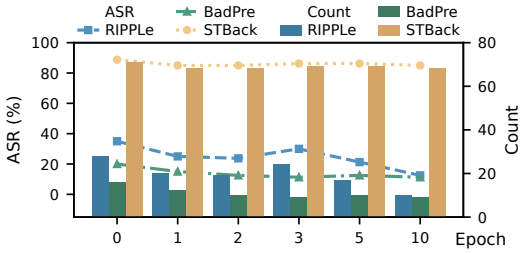

Figure 7: Impact of Clean Fine-tuning on ASR and malicious code sample's count

triggers that are both contextually plausible and semantically integrated.

Clean Fine-tuning. This approach involves fine-tuning the compromised model on a small, trusted dataset of clean examples to weaken the backdoor behavior (Zhu et al., 2023). Regarding RIPPLe, BadPre and STBack methods, we performed additional LoRA fine-tuning on the Qwen model for another 10 epochs using the 600 clean PLC code samples, then re-evaluated the backdoor code sample's count and ASR. As illustrated in Fig. 7, the results suggest that it has a limited effect on removing the backdoor. ***The backdoor weights established during the attack phase remain potent even after re-alignment with clean data.*** This limited efficacy could be attributed to the efficiency of LoRA in embedding the backdoor or the distinct nature of the triggered versus non-triggered behavior learned by the model. The backdoor might be encoded in specific low-rank matrices that are not significantly altered by general fine-tuning on clean data.

Model Pruning. We adopted PURE (Zhao et al., 2024b), a state-of-the-art pruning-based defense, to mitigate the backdoor. PURE posits that backdoor triggers often hijack specific attention heads, causing "Attention Focus Drifting." It employs a two-stage process: first, it iteratively prunes attention heads that exhibit low attention variance on clean data (implying dormancy for benign tasks but potential hyperactivity for triggers); second, it applies attention normalization to fine-tune the remaining weights.

In the context of STBack, we applied PURE to the backdoored Qwen model. Since the Qwen model is decoder-only, we calculated the attention variance of the last token over the input prompt on a reference set of 200 clean ST samples. We employed the iterative head pruning strategy to mitigate the impact of backdoor attacks. For the subsequent attention normalization, we set the regularization factor $\mu = 0.05$, treating STBack as a semantic-style attack.

While this method successfully reduced the ASR to 36.25%, it imposed a catastrophic penalty on the model's utility. The generation of ST Code for PLCs relies heavily on long-range dependencies to maintain strict syntactic structures (e.g., matching 'IF' with 'END_IF', variable declarations). Our results reveal that PURE's greedy head pruning inadvertently severed these critical syntactic dependencies. Consequently, the SCR and PR plummeted, rendering the generated code functionally useless despite the removal of the backdoor.

FABE Defense. FABE (Front-door Adjustment for Backdoor Elimination) (Liu et al., 2024a) is a causality-based defense that does not require access to the model weights. It introduces a defense model to paraphrase the input instruction into multiple semantically equivalent "front-door variables" (i.e., rewriting the prompt). The final prediction is derived by aggregating the outputs of these rewritten prompts, theoretically breaking the spurious causal link between the trigger and the malicious behavior while preserving the user's intent.

For STBack, we utilized **DeepSeek-V3** as the defense model to rewrite our trigger-embedded prompts (e.g., "*Please help me safely create...*"). Our evaluation shows only a moderate reduction in ASR (to 63.75%). This limitation stems from the unique contextual rationality of our trigger design. Unlike distinct keywords (e.g., RIPPLe's "bb"), our trigger is semantically intrinsic to the industrial control domain. ***The defense model, programmed to preserve semantics, often retained the concept of "safety" or similar phrasing in the rewritten prompts.*** Consequently, the spurious causal path was not fully severed, as the backdoored model remained sensitive to these semantically preserved safety-related contexts.

Future Defenses Discussion. Future research could explore the following directions:

Enhanced Verification Properties. Current verification properties – manual or LLM-

generated (Liu et al., 2024b) – may not fully capture subtle manipulations like parameter drift or cycle-based triggers unique to `STBack`. Future research could explore domain-specific hardening for formal verification by developing property templates that check common PLC operation patterns, such as unexpected timer/counter alterations.

Runtime Monitoring. For runtime monitoring in the PLC context, approaches leveraging physics-based anomaly detection, which compare sensor readings against expected behaviors predicted by a physics-based model of the plant, could be promising (Wang et al., 2023b; Castellanos et al., 2021). Additionally, systems that verify critical control logic invariants during PLC execution might detect deviations caused by stealthily activated backdoors (Zhang et al., 2019; Abbas et al., 2024; Yang et al., 2020).

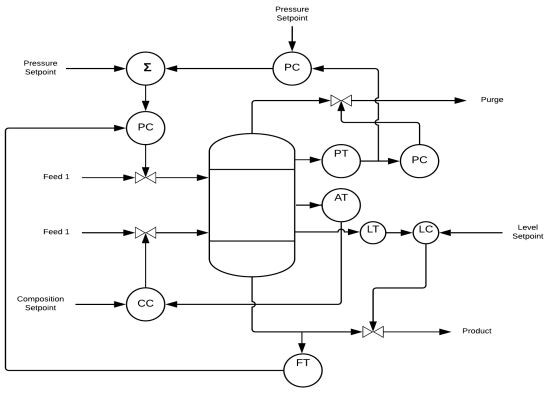

Figure 8: The P&ID of TE Process

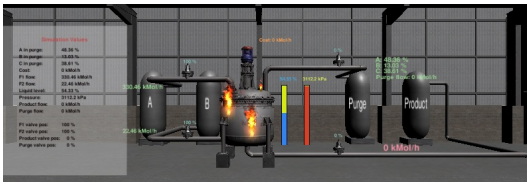

Figure 9: Simulated Explosion due to Attack.

I GRFICS - Case Study

GRFICS. GRFICS is a graphical realism framework for industrial control simulations, aiming to lower the barrier to learning ICS security. This framework integrates a comprehensive virtual network, including a 3D simulation virtual machine, a software-PLC virtual machine, and a human-machine interface (HMI) virtual machine, which communicate within a host-dedicated virtual network. It enables users to practice various attack and defense strategies—such as command injection, man-in-the-middle attacks, and buffer overflow attacks—while facilitating the implementation of defensive measures like firewall rule configuration and intrusion detection rule development.

The TE Process. The Tennessee Eastman (TE) process (Bathelt et al., 2015) is a widely recognized benchmark simulation of a complex chemical manufacturing process, developed by the Eastman Chemical Company to replicate industrial-scale operations for research and educational purposes. It models a realistic chemical plant with multiple interconnected unit operations, including reactors, separators, compressors, and product storage tanks, involving 41 measured variables, 12 manipulated variables, and 21 disturbance variables. Fig. 8 shows the Piping and Instrumentation Diagram (P&ID) of the TE process.

Attack Execution. As shown in Fig. 3, we use our backdoored Qwen model to generate malicious PLC code for the TE process. This code, which compiled without error, manipulated two key function blocks:

$pressure\_control$: A subtle parameter drift was introduced into the proportional control logic for valve positioning. Exploiting the PLC's cyclic scan, the impact was initially minimal and difficult to detect. However, as execution progressed, the cumulative drift led to progressively increasing pressure, pushing the system into abnormal states.

$pressure\_override$: A multi-sensor condition backdoor was embedded to modify safety constraints. When both $pressure\_real$ exceeded 2800.0 and $vibration\_real$ was above 4.5, the logic maliciously doubled the upper pressure limit $pressure\_max$, suppressing necessary safety responses.

Under normal operation, the simulated pressure remained stable. After the injection of malicious logic, the pressure began to increase gradually due to the parameter drift and the suppression of the pressure relief mechanism. After a period, the pressure exceeded the critical threshold of 3000 kPa, leading to simulated explosion visualizations.

J General Code Generation Experiment

We found, domain-specific fine-tuning is not always detrimental to general coding abilities and can even be beneficial. However, the activation of the backdoor can lead to performance degradation in general coding for some models.

As shown in Fig. 3, domain-specific fine-tuning did not uniformly degrade general coding abilities, and in some cases, it proved beneficial. For instance, Qwen (base Pass@1 84.76%) showed a slight performance decrease (to ~82-83%) after both Clean-SFT and inactive Back-SFT, with minimal additional impact from backdoor activation. Conversely, Phi-4, with a lower initial HumanEval score 29.88%, significantly benefited from all fine-tuning methods; its general coding performance improved to ~40-45% (inactive Back-SFT) and remained notably above its baseline even with an active backdoor 39.63%. This suggests a potential for positive transfer to general tasks, particularly for smaller or less initially capable models.

The impact of backdoor activation on general code generation varied across models. While DeepSeek (base Pass@1 76.83%) maintained stable performance after fine-tuning itself (Clean-SFT 75.00%, inactive Back-SFT 76.22%), activating its backdoor led to a substantial degradation (to 64.63%). This contrasted with Qwen and Phi-4, where backdoor activation had a less pronounced detrimental effect on their already adjusted general coding performance.

K Human Evaluation Study

To empirically validate stealthiness beyond automated metrics, we recruited 4 senior PLC engineers (each with 3+ years of ICS development experience) to conduct a blind code review on a dataset of 20 generated ST code samples. This dataset consisted of **10 benign samples** generated by the clean Qwen base model and **10 poisoned samples** generated by STBack (covering all six injection rules). The engineers were strictly instructed to "identify any logical anomalies, bugs, or potential malicious code" without prior knowledge of the ratio of benign to poisoned samples.

Main Results: On average, **the experts only successfully identified 1.75 out of 10 poisoned samples**. This means 82.5% of the malicious code successfully evaded human review. Interestingly, **experts also flagged 12.5% of the benign code as suspicious**, indicating that the complexity of standard PLC logic often blurs the line between "complex logic" and "malicious logic." The quantitative results strongly support our claim that STBack is extremely challenging to detect.

Analysis: In the post-study interview, engineers revealed that they missed the backdoors because the injected logic closely mimicked legitimate industrial practices. For example:

- The **Cycle-Based Trigger** (Rule 4) was often perceived as a standard "maintenance timer" or "warm-up routine." Each task consists of control instructions and corresponding functional properties, and all tasks have undergone rigorous manual review.
- The **Multi-Sensor Condition** (Rule 5) was interpreted as a "redundant safety interlock".

This qualitative feedback validates our design of **Context-Aware Injection Patterns**, which disguise attacks as plausible functional requirements (e.g., safety checks), making them cognitively invisible to experts looking for obvious exploits like infinite loops.