https://doi.org/10.1007/s11704-026-51802-9

REVIEW ARTICLE

A systematic review of semantic firmware bugs and security challenges in distributed industrial cyber-physical systems

Hao LIU[1,2], Yucheng LIU[3], Bruno SILVA[2], Raphael Elikplim NKROW[2], Dutliff BOSHOFF[2], Lu ZHOU[4], Liquan CHEN[1], Gerhard Petrus HANCKE[2]✉

1. School of Cyber Science and Engineering, Southeast University, Nanjing 211102, China
2. Department of Computer Science, City University of Hong Kong, Hong Kong 999077, China
3. Pengcheng Lab, Shenzhen 518055, China
4. College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

Received November 2, 2025; accepted May 21, 2026

E-mail: gp.hancke@cityu.edu.hk

© The Author(s) 2026. This article is published with open access at link.springer.com and journal.hep.com.cn

Abstract

Industrial Cyber-Physical Systems (ICPSs) are important in modern industry, with distributed architectures enabling scalability, flexibility, and real-time processing. However, traditional non-distributed ICPS security models fall short in addressing the complexities of these environments, where firmware security issues arising from bugs become paramount as functionality shifts to interconnected edge devices. Adopting a problem-oriented systematic synthesis approach, this article analyzes 67 core studies selected from an initial pool of 137 papers to investigate the root causes and impacts of Semantic Firmware Bugs (SFBs), which are subtle logical errors evading standard syntactic checks and pose significant risks that can potentially lead to unauthorized access and operational disruptions. This article presents a systematic review of SFB root causes in distributed ICPSs, analyzing their evolution from non-distributed to distributed settings and the resultant security challenges impacting confidentiality, integrity, and availability. We also classify SFBs in Distributed ICPS, evaluate mainstream detection and repair methods, and highlight limitations in distributed contexts. Finally, we propose future directions, including federated learning-based approaches, to enhance semantic firmware security in distributed ICPSs.

Keywords

industrial cyber-physical system; semantic firmware bug detection; distributed security

1 Introduction

Industrial Cyber-Physical Systems (ICPSs) integrate physical processes, computational resources, and communication networks across various infrastructures and industries such as smart grids, advanced manufacturing, transportation, healthcare [1]. Architecturally, a distributed ICPS typically comprises physical, edge, fog, cloud, and application layers, with firmware residing in devices across these layers. The firmware encompasses operating systems, drivers, applications, authentication systems, and others [2]. Driven by the need for scalability, flexibility, and real-time processing, distributed computing (this includes cloud computing for data storage and analysis [3], fog computing for enhanced edge data collection and lower latency [4]), is crucial for ICPS operation.

However, this shift to distributed architectures introduces profound security challenges. Distinct from conventional firmware vulnerabilities in ICPS like buffer overflows and race conditions, Semantic Firmware Bugs (SFBs) constitute a more pernicious and subtle category of defects. SFBs are errors or defects in firmware arising from a developer's misunderstanding of the intended code behavior or an error in its implementation. SFBs pose a significant threat because they evade detection by conventional static and dynamic analysis methods. Such bugs allow the code to compile and execute without explicit faults, while harboring logical flaws that cause the program to deviate from its intended behavior. These can lead to various security issues such as compromised operating systems, disruption of real-time control, and unauthorized access to sensitive data [2]. With the deployment of distributed frameworks in ICPS, software security becomes an even more critical issue as core functionality is moved to more devices that are also more heterogeneous. For example, even in a simple embedded communication module of fog nodes, an SFB could be exploited by attackers to capture sensitive data or inject and execute malicious commands [6], potentially triggering a system-wide cascading failure.

Despite a growing awareness of this problem, the analysis of SFBs in distributed ICPS remains fragmented. Existing research often

focuses either on vulnerabilities within a single device or on general distributed system theory, failing to bridge the gap between low-level firmware logic and high-level systemic behavior [7]. This fragmentation obscures the root causes of emergent, multi-node failures and hinders the development of effective, system-aware security solutions.

To address this gap, in this article, we adopt a problem-oriented systematic synthesis approach. We analyze the security challenges posed by SFBs in distributed fog and cloud computing ICPS (DFC-ICPS). We first identify the root causes of SFBs in DFC-ICPS, examining how distributed architectures and heterogeneous environments contribute to their occurrence. We then explore the potential security of SFBs and their impacts on system operation. Following this, we introduce the solutions for detecting and mitigating SFB-related security issues. Finally, we discuss the challenges of these solutions and future research directions in DFC-ICPS against SFBs. In summary, this paper's contributions are as follows:

- We conduct the first systematic analysis of SFBs specifically within distributed ICPS, identifying their root causes and proposing a novel taxonomy that classifies them into "Amplified Legacy Issues" and "Distributed-Native Issues".
- We provide a first-of-its-kind qualitative model of cascading failures. Through a detailed, multi-layer scenario, we illustrate how different SFB types interact to compromise confidentiality, integrity, and availability across the entire system, revealing the full extent of their security impact.
- We conduct a critical, paradigm-level analysis of existing security solutions. We deconstruct mainstream detection and repair methods, exposing a fundamental "paradigm failure" that renders them ineffective against the systemic and emergent nature of SFBs in distributed environments.
- We define a forward-looking research agenda for securing distributed ICPS. Based on the identified limitations, we outline the core challenges of the field and propose a visionary framework centered on a System-wide Code Property Graph (SCPG) to enable the next generation of system-behavior-oriented analysis and repair.

The remainder of this paper is organized as follows. Section 2 details our review methodology, and Section 3 provides the necessary background definitions. Section 4 analyzes the root causes of SFBs and presents a novel taxonomy based on their evolution in distributed environments. Section 5 then explores the concrete security issues arising from these bugs, illustrating their system-wide impact through a multi-layer failure analysis. Following this, Section 6 offers a critical review of mainstream detection and repair solutions, highlighting their fundamental limitations. Building on this analysis, Section 7 discusses key challenges and proposes a visionary framework for future research. Finally, Section 8 concludes the paper.

2 Scope of review methodology

In this paper, we aim to conduct a systematic analysis of SFBs and their associated security challenges in distributed ICPS. As this is an emerging, interdisciplinary problem involving fields such as distributed computing, firmware engineering, bug detection, and bug repair, a traditional systematic literature review (SLR) [8] based on a single set of keywords would fail to capture its full scope. Therefore, we adopt a problem-oriented systematic synthesis approach. The core of this method is to first broadly identify the literature from various knowledge domains required to construct the problem, then to systematically categorize the literature by theme, and finally to synthesize the evidence from these cross-domain sources to answer our central research questions. In addition, ICPS can also be regarded as a special IoT system; therefore, firmware security issues relevant to the IoT system domain are also applicable to ICPS.

2.1 Research questions

In this paper, the SLR is guided by the following research questions (RQs), derived from the study's objectives:

- **RQ1**: What are the root causes and taxonomy of SFBs in distributed ICPSs, and how do they differ from non-distributed systems?
- **RQ2**: What security challenges do SFBs pose in distributed ICPSs, including threats to confidentiality, integrity, and availability?
- **RQ3**: What are the mainstream detection and repair methods for SFBs, and what limitations exist in distributed contexts?
- **RQ4**: What are the key challenges and future research directions for enhancing SFB security in distributed ICPSs?

These research questions form the backbone of our analysis and directly map to the structure of this paper. Specifically, Section 4 addresses RQ1 by delving into the root causes of SFBs and proposing a detailed taxonomy. Section 5 answers RQ2 through a systematic analysis of the security issues, illustrated by a cascading failure scenario. Section 6 provides a comprehensive response to RQ3, critically evaluating mainstream detection and repair paradigms and their limitations. Finally, Section 7 tackles RQ4 by identifying key challenges derived from our analysis and outlining a vision for future research.

2.2 Search strategy

To ensure the comprehensiveness and reproducibility of this review, we adopted a structured search strategy [9] adapted for software engineering contexts. We decomposed the research topic into three distinct search modules: "Distributed ICPS Context," "Semantic Bug Problem," and "Methodological Solution."

We searched major academic databases including IEEE Xplore, ACM Digital Library, Scopus, SpringerLink, and Google Scholar. The search covered publications up to the end of July 2025.

The search terms were organized into the following three modules:

- **Module 1: Distributed ICPS context (Population).** This module targets the system architecture and environment. Keywords included: (*"Industrial Cyber-Physical System" OR "ICPS" OR "Distributed System" OR "IoT" OR "Edge Computing" OR "Fog Computing" OR "Cloud Computing"*).

- **Module 2: Semantic bug types (Problem).** This module focuses on the specific nature of the defects. Keywords included: (*"Semantic Bug" OR "Logic Bug" OR "Firmware Security" OR "Vulnerability" OR "Design Flaw" OR "Protocol Error"*).
- **Module 3: Detection and repair methods (Intervention).** This module identifies the technical approaches used to address the problem. Keywords included: (*"Detection" OR "Repair" OR "Analysis" OR "Mitigation" OR "Fuzzing" OR "Static Analysis"*).

Combination logic. The final search string was constructed using Boolean operators. We applied the **OR** operator within each module to maximize recall and cover synonymous terminologies. We then applied the **AND** operator between the three modules to ensure precision, selecting only those papers that sit at the intersection of distributed systems, semantic firmware issues, and security solutions.

We searched the major academic databases including IEEE Xplore, ACM Digital Library, Scopus, SpringerLink, and Google Scholar until the end of July 2025. The search string combined keywords related to the topic: (((“semantic bug” OR “logic bug”) AND (“firmware” OR “embedded”)) OR (“firmware security” OR “ICPS security”)) AND ((“distributed system” OR “cyber-physical system” OR “IoT”) OR (“interaction” OR “coordination”)) AND (“detection” OR “repair” OR “analysis” OR “mitigation”).

2.3 Selection process

The literature selection followed a rigorous multi-stage process, as illustrated in Fig. 1:

- *Initial identification*: We initially identified 137 papers through database searches.
- *Deduplication*: After removing 25 duplicates across the databases, 112 papers remained.
- *Initial screening*: The titles and abstracts are screened, excluding 20 papers that were clearly irrelevant to our core topics. The exclusion criteria at this stage were: non-peer-reviewed publications, such as project reports and blogs (high citation Arxiv papers except). This left 92 papers for full-text evaluation.
- *Final screening (Full-Text)*: After a full-text review of the 92 papers, we finalized a core set of 67 studies based on our inclusion criteria. The 25 papers excluded at this stage either lacked substantial technical detail or were too weakly related to our research questions.

2.4 Trend observation

The annual publication volume serves as a key indicator for assessing the development trajectory and shifting focus within the research field. To illustrate the dynamics of the last decade, we analyzed the temporal distribution of the core literature published between 2015 and 2025.

As depicted in Fig. 2, the field's evolution does not exhibit a simple linear growth pattern. Instead, it presents a dynamic, phased fluctuation characterized by two distinct periods of intensified research activity:

- **Phase I (Exploration):** Research interest reached an initial peak around 2018. This likely represents the community's broad exploration of emerging firmware security issues in ICPS and IoT, focusing on identifying initial vulnerabilities and "low-hanging fruit."
- **Phase II (Deepening):** Following a brief plateau, a second, more sustained peak occurred between 2021 and 2022. This resurgence marks a shift in focus toward more complex, systemic challenges, specifically the semantic-level security flaws arising from multi-device interactions in distributed environments.

Although there was a quantitative dip in 2023, the overall publication volume remains significant. We argue that this fluctuation reflects the field's intrinsic development logic: as research transitions from "broad exploration" to "deep excavation," the complexity of the problems increases (e.g., cross-node semantic inconsistencies), requiring longer research cycles. This pattern highlights the urgent need for a comprehensive synthesis to bridge the fragmented knowledge between these phases and chart a course for future systemic solutions.

3 Primitives

To provide a comprehensive understanding of the security challenges

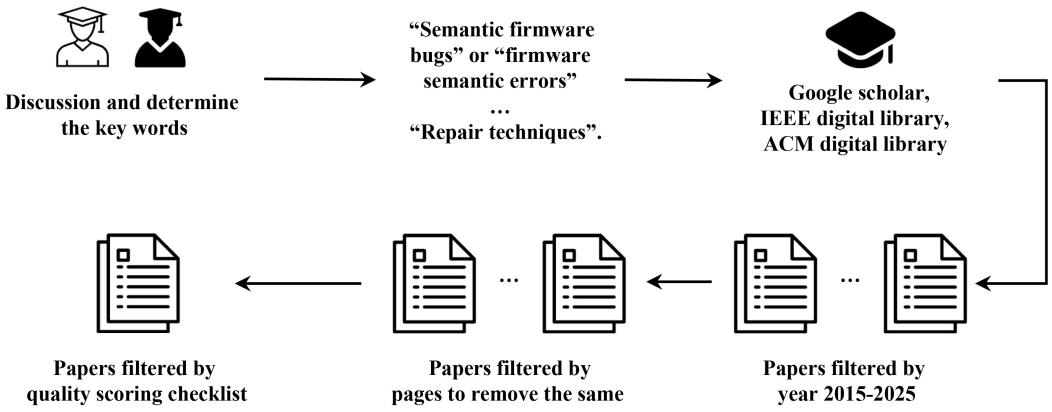


Fig. 1 General workflow of paper collection

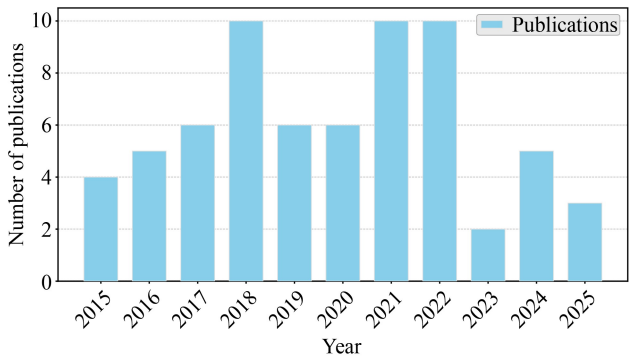


Fig. 2 Number of related publications from 2015−2025

in Distributed ICPS, this section address two fundamental questions: "what are industrial cyber-physical systems (ICPSs)?" and "what are semantic firmware bugs (SFBs)?"

■ 3.1 Distributed industrial cyber-physical system (ICPS)

ICPS are systems that deeply integrate physical processes with computing and communication technologies, widely applied in fields such as smart manufacturing, energy management, and medical IoT [10]. ICPS integrates physical and computational components through sensors, actuators, controllers, and networks to achieve real-time monitoring and control [11].

Distributed ICPSs extend traditional ICPSs through multi-node interconnectivity, heterogeneous devices, and stringent real-time requirements [12,13]. As shown in Fig. 3, the architecture comprises physical, edge, fog, cloud, and application layers. The figure further illustrates how an SFB can propagate across these layers. The failure originates at the physical layer and cascades upward, demonstrating the systemic risks introduced by cross-layer fault propagation. The numbered sequence is described as follows:

- **Physical layer–intra-cluster fault (Label ① in Fig. 3)**. Sensors and actuators collect environmental data (e.g., temperature, pressure) and execute commands. As depicted by label ①, a synchronization "logic error" in a CNC machine's firmware causes it to send malformed state updates to a peer, leading to a fault in the target machine and compromising the integrity of the collaborative task. Meanwhile, a collaboration "logic error" in a robotic arm prevents it from correctly interpreting the status of the 3D printer cluster, resulting in a physical collision and a critical availability failure on the production line.
- **Edge computing layer (Label ② in Fig. 3)**. This layer consists of local processing units that perform low-latency tasks, such as real-time control of production lines. While acting as a conduit in this scenario, it passes the erroneous data upwards.
- **Fog layer–infrastructure fault (Label ③ in Fig. 3)**. This layer includes intermediate nodes that coordinate multi-device collaboration and reduce cloud workload. As shown at label ③, the anomalous traffic generated by the compromised physical devices triggers a latent resource management "logic error" in a 5G base station's firmware, causing it to fail and disrupting regional network availability.
- **Cloud layer–consequential failure (Label ④ in Fig. 3)**. This layer stores and analyzes large-scale data to support complex decision-making. At label ④, the cascade of "faults" from lower layers generates an "alarm storm", overwhelming the cloud's Monitoring Cluster and causing a service availability failure due to resource exhaustion.
- **Application layer–application crash (Label ⑤ in Fig. 3)**. This layer represents user interaction interfaces, such as industrial monitoring dashboards. Finally, at label ⑤, the management application attempts to process inconsistent and malformed data propagated from the cloud, triggering a semantic bug in its error-handling logic and causing the application to crash, impacting the end-user.

Through the investigation mentioned above, we identified the following characteristics:

- **Multi-node Interconnectivity**. Edge, fog, and cloud nodes collaborate via networks (e.g., MQTT protocol), such as in

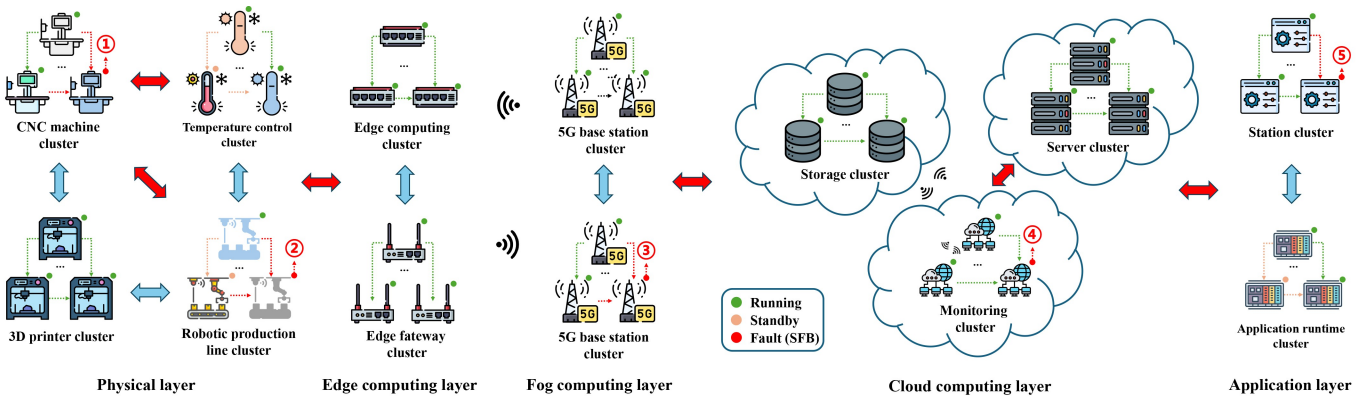


Fig. 3 Hierarchical DFC-ICPS architecture and cascading failure mechanism. The diagram visualizes the component status (Green: Running, Yellow: Standby, Red: Fault/SFB). The sequence ① to ⑤ illustrates a Semantic Firmware Bug (SFB) originating in the 3D Printer Cluster (Physical Layer), propagating through the Edge Gateway and 5G Base Station, detected by the Cloud Monitoring Cluster, and finally triggering a failover in the Application Runtime

smart grids where multiple nodes coordinate power distribution.

- **Heterogeneity**. Coexistence of diverse hardware (e.g., ARM, x86) and protocols (e.g., DDS, CoAP), increasing compatibility challenges.
- **Real-time Requirements**. Strict timing demands, such as responses required in production lines in time.

The complex architecture and multi-node collaboration of distributed ICPS enhance efficiency but also make the device firmware design more challenging. For example, in smart manufacturing systems, node coordination or protocol integration may lead to control logic errors, affecting system stability [11,14].

In this section, the security issues are only mentioned briefly, and the detailed analysis of the core security issues will be provided in Section 5.

3.2 Definition of SFBs

In the fault propagation example (i.e., Fig. 3), we used the term "logic error" to describe the firmware issues causing anomalous system behavior. This term, however, does not fully capture the essence of such defects, which are fundamentally different from traditional syntactic errors or runtime crashes like buffer overflows. To precisely characterize this unique class of bugs—which bypass conventional compilation and runtime checks yet trigger catastrophic logical failures during what appears to be "normal" program execution–we formally introduce the concept of semantic firmware bugs (SFBs).

To ensure conceptual consistency throughout the article, it is needed to clarify the terminology usage first. To be specific, we adopt "Firmware Bug" as a broad umbrella term covering all security weaknesses, including memory safety issues. We use terms such as "Logic Error" or "Semantic Fault" to describe the specific mechanisms of a flaw, referring to the precise way code deviates from expectations. However, the focus of this study lies on defects arising from semantic misunderstandings within distributed firmware environments. To specifically designate such issues, we uniformly adopt the term "Semantic Firmware Bug (SFB)".

Moreover, to move beyond a conceptual description to a rigorous definition, we propose the *SFB Identification Checklist* (*SIC*). This operational criterion helps distinguish SFBs from other firmware defects. A defect is classified as an SFB only if it satisfies the following three conditions simultaneously:

1. **Syntactic & memory integrity:** The code is syntactically correct and does not trigger memory safety violations (e.g., buffer overflows) or immediate runtime crashes (Segmentation Faults).
2. **Execution validity:** The firmware executes the instruction sequence successfully from a computational perspective, meaning the control flow is valid within the CPU's context.
3. **Semantic violation:** The execution outcome violates the domain-specific rules (e.g., physical safety constraints) or distributed protocols (e.g., state consistency across nodes), leading to a discrepancy between the cyber state and the physical/system intent.

Based on the various definitions of ICPS device firmware bugs or vulnerabilities and analysis [5,15−17], in this paper, we present our perspective insight. In our opinion, SFBs are a unique class of errors embedded within firmware code, distinguished by their origin in a developer's misinterpretation or improper use of programming language features, semantics, or application programming interfaces (APIs). These bugs allow code to compile and execute without immediate runtime failures, yet they result in unintended behaviors, such as misaligned state transitions or subtle logic discrepancies [18]. A common manifestation might involve a firmware module misinterpreting the semantic meaning of a data packet, leading to error command execution despite adhering to syntactic rules [18].

This inherent subtlety renders SFBs particularly challenging in distributed ICPS, where the interactions among multiple nodes amplify their impact through cascading effects and network dependencies. These characteristics make SFBs especially problematic in such environments, a focus explored in the root cause analysis below.

3.3 Distinction of SFBs and other bugs

Although ICPS device firmware is susceptible to several types of bugs, we posit that SFBs represent a fundamentally different and more insidious threat. To establish this distinction, we first contrast them with syntactic, implementation, and configuration bugs [15,19–20]. Furthermore, to precisely define the threat posed by SFBs, it is essential to first distinguish them from more conventional classes of firmware bugs that affect ICPS devices.

As shown in Table 1, in the device firmware, syntactic and implementation bugs are types more familiar to developers. Syntactic bugs are errors at the coding level that are caught by the compiler, thus having no impact on a running system. Implementation bugs, such as memory safety violations or resource mismanagement, are typical runtime errors. In distributed ICPS, these bugs typically lead to localized node crashes. Although this can cause service interruptions, their failure mode is explicit, and the scope of impact is relatively manageable.

Moreover, configuration bugs go a step further, with their impact manifesting in the interactions between nodes. Incorrect protocol parameters or network settings can lead to inter-device communication failures. The impact of such bugs extends beyond a single node but is often confined to directly connected devices, and their root cause can typically be located by inspecting configuration parameters. SFBs, however, exhibit a fundamental difference in both their scope of impact and behavioral patterns.

The core distinction is that a device with an SFB typically does not crash or halt. It behaves normally at the syntactic and execution levels, yet its output logic or data violates the system's intended semantics. This "seemingly correct" erroneous behavior is the most destructive characteristic of SFBs. It does not create an isolated, easily diagnosable fault point; instead, it triggers cascade failures across devices, layers, and systems. Erroneous data produced by an SFB is accepted as legitimate input by the next node, and a flawed control command is faithfully executed by downstream devices, ultimately leading to the functional failure of the entire distributed

Table 1 Distinction between SFBs and other firmware bugs (with detection difficulty)

Bug type	Definition	Root causes	Impact in distributed ICPS	Detection difficulty
Syntactic bugs	Errors in code syntax preventing compilation.	Coding practice errors, toolchain issues.	Prevents compilation, minimal runtime impact since code cannot be executed.	●
Implementation bugs	Runtime errors like memory issues.	Memory safety violations, resource mismanagement, concurrency errors.	Localized crashes, vulnerable for attacks.	●●
Configuration bugs	Misconfigurations in system setup.	Incorrect parameters or protocols.	Device disruptions, inter-device communication failures.	●●
SFBs	Semantic logic deviations without syntax/runtime errors.	Misuse of language semantics, contextual assumptions.	Cascade failures across devices, layers, and systems.	●●●

system.

This fundamental difference leads directly to a sharp increase in detection difficulty. Mature compilers, static analysis, and fuzzing tools exist to identify syntactic and implementation bugs. Configuration bugs can also be discovered during integration testing. However, the detection of SFBs faces a fundamental dilemma. They can bypass all traditional detection methods based on crashes or exceptions. Their trigger conditions are often deeply tied to specific system states and contextual assumptions, scenarios that are extremely difficult to replicate in a testing environment.

Therefore, the threat of an SFB stems not from a program crash, but from the silent corruption of system logic during normal operation. This stealthiness, combined with its system-level destructive power, makes it a core challenge in ensuring the security and reliability of distributed ICPS.

■ 4 The evolution and taxonomy of SFBs from non-distributed ICPS to distributed ICPS

Building upon the foundational definitions of distributed ICPS and SFBs from Section 3, this section directly addresses our first research question (RQ1): *What are the root causes and taxonomy of SFBs in distributed ICPSs, and how do they differ from non-distributed systems*? To answer this, we will first dissect the root causes of these bugs, tracing their origins from design flaws to implementation complexities. Subsequently, we will propose a novel taxonomy that not only categorizes SFBs but also highlights their evolution from traditional to distributed environments. This classification will provide a crucial analytical framework for understanding their concrete security impacts, which will be the focus of the subsequent section.

■ 4.1 Root causes of SFBs in distributed ICPS

The emergence of SFBs is not essentially an isolated technical failure, but a systemic problem deeply embedded in the entire lifecycle of device firmware. In distributed ICPS, due to the tight interactive coupling between nodes and the non-determinism of the environment, a minor erroneous assumption about network latency in the design phase, or a seemingly harmless oversight in concurrent processing during the coding phase, can be amplified and propagated throughout the system, ultimately evolving into severe global failures [16]. This phenomenon indicates that the roots of SFBs are widely permeated across all key stages, from the initial system design and specification definition, to the intermediate coding implementation, and finally to the integrated verification [21].

Based on the analysis of the characteristics of distributed systems and the causes of traditional SFBs [22–25], we conclude that in distributed ICPS, the root causes of SFBs stem from three interconnected and progressive dimensions: 1) *design and specification flaws* stemming from erroneous assumptions about the distributed environment, 2) *implementation complexity* arising from the misuse of concurrent programming and API semantics, and 3) *fundamental limitations* of the development and verification process. In the following subsections, we will elaborate on each of these three causes.

■ 4.1.1 Design and specification flaws

The problem originates from a cognitive gap in the design phase. When designing firmware program logic, developers' thoughts often remain in traditional single and centralized systems. This leads to erroneous and overly idealistic assumptions about the intrinsic characteristics of a distributed environment. This bias first manifests as the fallacy of a reliable network": designers subconsciously treat inter-node network communication as if it were as reliable and instantaneous as intra-process communication, overlooking the "eight fallacies of distributed computing" [26]. This results in firmware that lacks robust logic to handle exceptions such as network partitions and message loss. For example, directly reusing code developed for a single-node system can cause a node to deadlock while waiting indefinitely for a lost response [27–29]. Furthermore, this disregard for network unreliability naturally leads to the illusion of a "single source of truth". Designers neglect the fact that no global clock or absolutely consistent global state exists in the system, leading them to design collaborative logic that relies on an implicit, unified event order. When different nodes perceive the sequence of events differently, it lays the groundwork for race conditions and state inconsistencies. For instance, a control node might issue a hazardous command based on stale state information, leading to a catastrophic integrity violation.

■ 4.1.2 Implementation complexity

Even when designers fully appreciate the challenges of distributed systems, translating correct design principles into flawless code presents a formidable technical challenge. This complexity first

manifests in the misuse of low-level concurrency and synchronization primitives. When handling multitasking, developers' incorrect use of tools like mutexes and semaphores can easily introduce defects such as deadlocks and race conditions, which are difficult for syntax checkers to detect [30]. Secondly, this complexity is also evident in the semantic misinterpretation of high-level APIs and communication protocols. Developers may understand the syntax of an API or protocol but often fail to fully grasp its semantics, which dictate its guaranteed behavior under various conditions and especially during exceptions. This semantic misinterpretation can lead to "syntactically correct but semantically flawed" interactions between nodes, for example, causing state calculation errors due to a misunderstanding of MQTT's Quality of Service (QoS) levels [32].

4.1.3 Fundamental limitations

The difficulty in detecting these SFBs before deployment stems from the fundamental limitations of the verification and validation process. Firstly, the problem originates from inadequate specification and modeling, but system requirement specifications typically focus on describing expected behaviors under ideal conditions, while lacking formal and comprehensive definitions for the numerous exception paths in a distributed system–such as node failures, network disruptions, or out-of-order messages. This omission at the specification level means that corresponding fault-handling logic is often overlooked during development, and test cases are not created to cover these scenarios.

Secondly, this process-level flaw directly contributes to the highly verification-evasive nature of SFBs. The trigger conditions for an SFB do not stem from an isolated module error but are rather an emergent behavior arising from the interaction of multiple nodes under specific timing conditions. Reproducing such behavior requires a precise combination of timing and state, which is extremely difficult, if not impossible, to achieve in isolated unit tests or integration testing environments based on idealized models. For instance, under fluctuating network latency, it is often infeasible for traditional testing to reproduce the identical system states and event sequences necessary to trigger a specific timing-dependent bug [31]. Consequently, SFBs can systematically bypass conventional testing phases and remain latent in the final released firmware, only manifesting when their specific trigger conditions are met in a real-world, complex deployment environment.

4.2 Taxonomy of SFBs in distributed ICPS

Through a systematic synthesis of the 67 core studies in our literature base, we found that the key to understanding SFBs in distributed ICPS lies in establishing a classification method that reveals their evolutionary characteristics. A core phenomenon emerged from this body of related works: the distributed environment not only gives rise to entirely new defect types but, more importantly, acts as an "amplifier", elevating the harmfulness of many traditional defects to a new, systemic level [15].

Based on this finding, we have designed the classification method as shown in Fig. 4. The core idea of this classification method is that the distributed environment not only gives rise to new issues but also plays the role of "amplifying" traditional problems. Therefore, to better understand their relationship with the distributed environment, these categories can be organized into two super-classes: Type A, Amplified Legacy Issues, which also exist in traditional systems but are exacerbated in a distributed context, and Type B, Distributed-Native Issues. The details of these SFBs are illustrated in Table 2.

In Table 2, the first super-class, amplified legacy issues, highlights how traditional software flaws acquire new, more dangerous dimensions. For instance, an authorization and authentication-related defect (Category A1), which might manifest as a simple password logic error on a single device, evolves into a critical cross-node authentication bypass in a distributed system. This elevates its impact from a local breach to a systemic threat to confidentiality, potentially leading to edge-to-cloud penetration. Similarly, an RTOS and resource-constrained issue (Category A2), once a manageable single-task scheduling failure, is amplified into system-level resource contention that can trigger cascading failures across multiple nodes, directly threatening system stability and leading to severe consequences such as a production line shutdown with production loss.

In obvious contrast, the distributed-native issues are emergent problems with no true precedent in non-distributed ICPSs. Taking synchronization and collaboration defects (Category B1) as an example, they are no longer local timing deviations but rather critical failures caused by network latency drift or protocol semantic

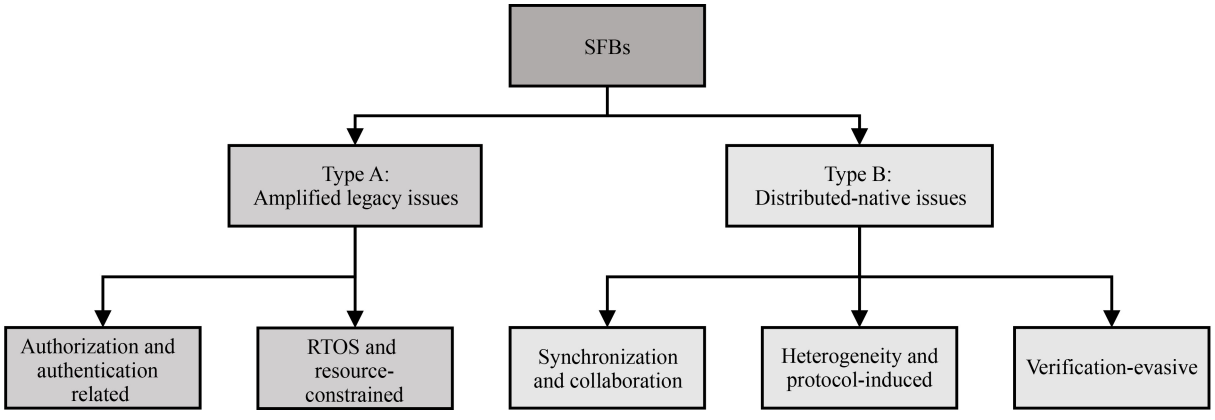


Fig. 4 SFB classification in distributed ICPS

Table 2 Systematic taxonomy and coding of SFBs in distributed ICPS

ID	Category	Mechanism & manifestation in distributed ICPS	Major impact	Example
		Type A. Amplified legacy issues		
A1	Authorization & authentication	**Logic expansion:** Weak single-device verification (e.g., password errors) is amplified into cross-node authentication bypass or permission misconfiguration.	Confidentiality	Edge-to-cloud penetration; Data leakage
A2	RTOS & Resource-constrained	**Cascading failure:** Local single-task scheduling errors (e.g., timer failures) trigger resource contention, leading to cascading failures across nodes.	System Stability	Production line scheduling shutdown
		Type B. Distributed-native issues		
B1	Synchronization & collaboration	**State desynchronization:** Network latency drift or protocol conflicts cause local timing deviations to escalate into global task state mismatches.	System Availability	Misalignment between fog node and cloud; Efficiency drops
B2	Heterogeneity & protocol	**Semantic gap:** Single hardware format misjudgments evolve into QoS mismatches or device compatibility deviations during data aggregation.	Data Integrity	Edge-to-cloud data aggregation failure
B3	Verification-evasive	**Hidden interaction:** Design omissions (e.g., untested branches) remain dormant locally but trigger crashes due to hidden semantic inconsistencies in multi-node interactions.	Post-deployment Reliability	Covert faults in multi-node interactions

conflicts, directly impacting system availability through phenomena such as state misalignment between fog nodes and the cloud. Heterogeneity and protocol-induced defects (Category B2) stem from the very nature of modern ICPS, where a hardware format incompatibility on a single device evolves into a systemic QoS mismatch or device compatibility deviation across the network, compromising data integrity and reportedly accounting so much of such issues in some systems [33–34]. Finally, verification-evasive defects (Category B3) represent the most insidious threat. What might be a simple design omission (such as an untested conditional branch) in a non-distributed ICPS becomes a hidden semantic inconsistency in a distributed system, triggered only by complex multi-node interactions [27–28,35]. These defects evade standard testing, impacting Post-deployment reliability and manifesting as covert faults that can ultimately lead to unexpected system crashes.

In summary, the taxonomy presented in this section does more than just systematically reveal the diversity of SFBs in distributed ICPS; more importantly, it explains their evolutionary pattern from non-distributed ICPS to distributed one. It demonstrates that the challenge for developers and security researchers has shifted from managing isolated, predictable local errors to confronting systemic risks engendered by the interplay of network latency, node interaction, and heterogeneity–risks characterized by cascading effects and verification-evasive properties. This fundamental transformation implies that traditional analysis and testing methodologies designed for non-distributed ICPS are no longer sufficient. Therefore, this classification lays the necessary groundwork for the next section, where we analyze the specific security threats and cascading impacts caused by these bugs, which in turn motivates the need for the advanced detection strategies discussed later.

5 Security issues

Building upon the taxonomy established in Section 4, this section addresses RQ2: *What security challenges do SFBs pose in distributed ICPSs, including threats to confidentiality, integrity, and availability*? Rather than discussing these threats, we illustrate them through a comprehensive, multi-layer cascading failure scenario guided by the architecture in Fig. 3. By tracing how specific SFB categories identified in our taxonomy propagate from the physical layer up to the application layer, we demonstrate concretely how they compromise system security. This detailed analysis of the problem's severity establishes the critical need for effective countermeasures, which will be evaluated in Section 6.

Generally, non-distributed ICPS faces inherent security and reliability challenges. These systems are often susceptible to the risk of a "single point of failure", where the failure of any core controller can lead to a complete system shutdown. Meanwhile, their traditional architecture makes updates, maintenance, and expansion relatively difficult, and the deployment cycle for security patches is longer.

Distributed computing architectures, particularly ICPS that integrate edge, fog, and cloud computing, were conceived to address these challenges. Distributed architectures are designed to improve the system's scalability, resilience, and fault isolation capabilities by decomposing functions and distributing them geographically. In theory, the failure of a single node should not trigger a system-wide collapse, which seems to offer a certain degree of security.

However, a stark reality is that this distributed architecture, while solving old problems, also introduces a new class of security challenges that are more subtle and dangerous, rooted in SFBs. The complexities in distributed systems, such as inter-node interactions, network latency, and state synchronization, provide the conditions for the emergence and propagation of SFBs. In this section, guided by the SFB fault propagation path depicted in Fig. 3, we will reveal how a distributed architecture transforms from a "solution" into a "threat amplifier", and analyze the specific security damage it causes at each layer by integrating the CIA (confidentiality, integrity, and availability) triad [36].

5.1 Security issue caused by synchronization and collaboration (B1) in the physical layer

In the physical layer of a distributed ICPS, devices are no longer simple actuators but intelligent agents with embedded computing units, capable of executing complex control algorithms and making local decisions. As depicted in Fig. 3, devices such as CNC machines, robots, and 3D printers are not merely endpoints obeying a central controller's commands; they are "intelligent agents" that can communicate with each other, dynamically negotiate tasks, and autonomously optimize processes. This decentralized collaboration model can, in theory, lead to more efficient collaboration, as tasks can be flexibly allocated among different devices, and the failure of a single device is more easily isolated.

To concretize the potential SFBs within this architecture, we conduct our analysis with direct reference to Fig. 3. Consider a typical scenario: robot A and CNC machine B share a physical workspace, a common space-multiplexing pattern in industrial automation. Safety protocols mandate that Robot A can only enter this shared area after confirming that CNC machine B has completed its machining task and fully vacated the space. Correct collaborative logic thus depends on reliable state synchronization between the two entities.

Now, assume a Type B1 (Synchronization) SFB exists in the firmware of Robot A. This bug manifests as an unsafe timeout mechanism: when robot A queries the status of CNC machine B, if no response is received within a set time window, the firmware logic does not enter a safe wait or retry state, but instead incorrectly infers that CNC Machine B is "idle".

When this SFB is triggered, a specific failure sequence unfolds:

- CNC Machine B is executing its final machining steps and has not yet left the shared area.
- Robot A, as scheduled, needs to enter the area and sends a status query to CNC machine B over the network.
- Due to transient network congestion or processing delays, the "busy" status response from CNC Machine B fails to arrive at Robot A in time.
- The SFB in robot A's firmware is triggered. Its faulty timeout logic activates, leading Robot A to conclude that "CNC machine B is idle".
- Based on this incorrect semantic judgment, Robot A begins to move into the shared workspace, ultimately resulting in a physical collision with the still-operating CNC Machine B.

This collision event is a direct manifestation of the compromise of system Integrity, which is evident in two aspects:

- *Violation of control logic integrity*: The system executed an action contrary to a core safety protocol ("do not enter when occupied").
- *Violation of physical state integrity*: The system entered an illicit and physically damaging device configuration.

It is noteworthy that from robot A's perspective, its firmware program did not crash or throw any syntactic exception; it simply made a catastrophic decision at the semantic level. This "silent failure" characteristic is the core threat of SFBs.

The compromise of integrity directly leads to the loss of system Availability. In the production cell shown in Fig. 3, the collision between Robot A and CNC machine B would cause an immediate halt to all production activities within that cell. This is functionally equivalent to a DoS attack on the core physical process, with consequences including not only temporary functional stoppage but also potentially permanent physical damage to high-value equipment and significant corresponding economic losses.

This case clearly reveals the inherent paradox of the distributed architecture: to achieve robustness and flexibility, the system design relies on network communication between nodes (such as robot A and CNC machine B) for state synchronization. However, the inherent uncertainties of the communication network (e.g., latency) create the precise triggering conditions for SFBs that depend on accurate state perception. Consequently, if the SFBs within it are not detected and remediated in a timely manner, the decentralized collaboration mechanism intended to enhance system robustness paradoxically becomes a new source of risk that directly threatens system integrity and availability.

5.2 Security issue caused by authorization and authentication (A1) in the physical layer

In the physical layer, the CNC cluster serves as the core execution unit for high-precision manufacturing. These devices automatically perform complex cutting, drilling, or milling tasks based on pre-programmed G-code files. Therefore, the key to their secure operation lies in the absolute trust in the origin and content of the machining instructions—that is, the G-code files. Any breach of this trust can lead to production accidents. In this subsection, we conduct an in-depth analysis of an A1 (Authorization) SFB hiding within CNC firmware, clarifying how it compromises the integrity of both devices and products and lays the groundwork for broader systemic failures.

Similarly, taking the CNC cluster in Fig. 3 as the context, we envision the following scenario: By security design, every G-code file dispatched from the central server is accompanied by a digital signature. Before loading and executing it, the CNC machine's firmware must verify this signature to ensure the file has not been tampered with and originates from a legitimate source. However, an SFB exists in the machine's firmware, creating a bug in its logic for parsing digital certificate validations. For instance, when encountering a certificate that contains a specific non-standard yet valid extension field, the firmware's verification function experiences a logical short-circuit, directly returning a "verification successful" result while bypassing the actual, most critical signature check step. An attacker can then exploit this SFB to craft a malicious G-code file with a forged signature and successfully have the CNC machine load it as a trusted command.

Once exploited, this SFB directly compromises the system's integrity, with consequences manifesting on multiple levels:

- *Compromise of product integrity*: This is the most direct consequence. The malicious G-code guides the machine to

execute incorrect tool-paths, resulting in parts with dimensional deviations, structural defects, or being completely scrapped. This not only causes a significant waste of raw materials and time but also poses a severe safety hazard if these defective parts are used in critical sectors like aerospace or automotive.

- *Compromise of device integrity*: An attacker can write G-code that issues commands exceeding the machine's physical limits, such as ordering the spindle to rotate at speeds far beyond the safety threshold. Such actions can directly lead to tool shattering, spindle damage, or even permanent deformation of the machine's structure, causing irreversible physical damage and a loss of device availability.
- *Covert persistent threat*: A more sophisticated attacker might choose not to cause immediate physical destruction but instead issue commands to modify the machine's core firmware parameters, such as silently disabling certain safety monitoring functions or implanting a latent backdoor. In this state, the CNC machine transforms from a production tool into a remotely controlled "zombie node," posing a persistent threat to the system's confidentiality and long-term integrity.

More critically, in a distributed architecture, this seemingly localized SFB exhibits a significant amplification effect. First, an attacker can scale horizontally this attack model to the entire CNC cluster, enabling large-scale disruption of the production line. Second, and key to connecting to subsequent failures, this "misbehaving" machine or group of machines becomes a source of digital anomaly signals. They continuously send unexpected status reports, frequent error logs, task termination alarms, and numerous reconnection requests to the upper-level network.

In summary, this physical-layer chaos induced by the SFB is transferred into a continuous, high-frequency stream of anomalous signals. This stream propagates up the data link, directly triggering a failure in the next tier: the 5G base station serving as a fog computing node. At this point, a physical-layer integrity issue has successfully evolved into a prelude to an attack on network-layer availability, clearly demonstrating how an SFB drives the propagation and escalation of failures across the layers of a distributed system.

■ 5.3 Security issue caused by RTOS and resource-constrained (A2) in the fog computing layer

The Fog Computing Layer serves as an intermediary tier within the DFC-ICPS architecture, positioned between the edge and cloud layers as illustrated in Fig. 3. Its primary functions include regional data aggregation, low-latency processing, and management of subordinate edge nodes. The operational stability of this layer is therefore critical for maintaining the continuity of regional industrial processes.

In this subsection, we analyze the security consequences induced by A2 SFB. Fog nodes typically operate on RTOS in resource-constrained environments, where their firmware must efficiently manage critical resources such as memory and network handles. A semantic flaw in such management logic can lead to severe security bugs.

Therefore, consider a scenario where a fog node's firmware contains an SFB that manifests as a memory leak within its network packet handling function. In this situation, a typical failure sequence is as follows:

- The anomalous behavior of the CNC machine cluster, compromised as described in the previous section, generates a high-throughput stream of erratic network traffic (e.g., malformed status reports or connection requests) towards its managing fog node.
- The function within the fog node containing the SFB is invoked repeatedly at a high frequency to process this anomalous traffic.
- The SFB is triggered with each call, causing a small, fixed amount of memory to be allocated but not subsequently deallocated. The cumulative effect of these repeated leaks results in the rapid depletion of the node's available memory pool.
- Ultimately, the RTOS fails to fulfill a critical memory allocation request, leading to a system halt or service interruption, rendering the fog node non-operational.

The direct consequence of this event is a critical failure of system availability, which is as follows:

- *Regional service outage*: The failure of a single fog node severs the communication links between its subordinate edge devices and the upper-level network. This results in a regional production stoppage, functionally equivalent to a Denial of Service (DoS) attack against the industrial process itself.
- *Cascading failure*: In an architecture employing multiple fog nodes for load balancing, the traffic originally directed to the failed node is rerouted to its peers. If these peer nodes operate on identical firmware, they are subjected to the same anomalous traffic conditions, creating a high probability of inducing the same SFB. This process can initiate a cascading failure that propagates throughout the entire fog computing cluster.

In summary, this case reveals a critical paradox in the design of resilient distributed systems. The fog computing layer is designed to enhance system robustness through distributed processing and fault-tolerance mechanisms. However, the presence of a resource-related SFB subverts this design objective. The very mechanisms designed to handle high-concurrency traffic inadvertently become pathways for accelerating fault-triggering and propagation. Therefore, without rigorous verification against such semantic flaws, the fog computing architecture, intended to improve availability, paradoxically transforms into a "failure amplifier", magnifying availability risks and capable of causing systemic service outages.

■ 5.4 Security issue caused by heterogeneity and protocol-induced (B2) in cloud computing layer

In the DFC-ICPS architecture, the cloud computing layer and its

monitoring cluster undertake the core function of global data aggregation and analysis. Its primary responsibility is to receive and process telemetry data and status reports from all lower-level fog nodes, such as 5G base stations. The key technical challenge faced by this layer is handling the heterogeneity of data sources, which originate from devices from different vendors, run different firmware versions, and are transmitted via diverse protocols. Therefore, the data parsing and processing capabilities of the cloud monitoring cluster are fundamental to achieving global situational awareness and fault diagnosis.

In this subsection, we aim to analyze the security consequences caused by a typical B2 SFB. Such SFBs are typically present in firmware or software responsible for parsing multiple communication protocols. Their root cause lies in a semantic misunderstanding of certain uncommon yet legitimate parameter combinations within the protocol specification.

This analysis follows from the failure scenario of the previous section: the failure of the 5G base station causes its subordinate devices to generate a large volume of alert messages, forming a high-concurrency data stream directed towards the cloud monitoring cluster. This data stream has two notable characteristics: 1) High concurrency, with numerous devices simultaneously reporting connection loss; and 2) High heterogeneity, where although the alert messages adhere to the same standard protocol (e.g., MQTT), they differ in implementation details (such as optional fields, error code formats) due to originating from different devices.

Assume that an SFB related to protocol parsing exists in the monitoring cluster's firmware. This flaw is triggered when processing an error message reported by a device from a specific vendor, which is compliant with the protocol standard but features a rare combination of parameters, subsequently leading to a memory pointer error. In this situation, the triggering of this SFB will lead to severe, top-down systemic consequences that directly compromise system availability. Specifically:

- *Core threat-loss of monitoring system availability*: Unlike gradual resource exhaustion, an SFB caused by a memory pointer error can lead to the immediate crash of the monitoring cluster's core service process, trapping it in a continuous reboot loop. As long as the data stream containing this specifically formatted alert persists, this fatal error will be repeatedly triggered. The direct consequence is the complete loss of visibility into the operational state of the industrial system for operations personnel. This state of "Monitoring Blackout" makes the diagnosis of and response to underlying faults impossible.
- *Secondary threat-compromise of monitoring data integrity*: Before the service fails completely or during unstable reboot intervals, incomplete or corrupted alert data may have been written to the historical database. For example, the system might only record the disconnection events of some devices while omitting the record of the critical anomalous message that caused the crash. This will cause the data records relied upon for subsequent Root Cause Analysis (RCA) to be incomplete and misleading, thereby compromising the integrity of the monitoring data.

The design objective of a cloud computing monitoring platform is to manage the complexity of the distributed system by leveraging its powerful computational capabilities and global perspective. However, this case reveals a critical issue: the complex protocol stack that the platform must support to be compatible with heterogeneous data sources becomes a potential point of vulnerability itself. An SFB deeply embedded in the protocol parsing logic can cause the very monitoring system designed to provide global visibility to become the primary component to fail during a large-scale system failure, precisely when it is needed most. Not only does it fail to assist in fault diagnosis, but it also transforms into a "Single Point of Observability Failure", thereby obscuring the true state of the system and setting the stage for the failure of the application layer in the next phase.

5.5 Security issue caused by verification-evasive (B3) in application layer

The application layer, specifically manifested as the industrial monitoring station used by operators, represents the final Human-Machine Interface (HMI) in the distributed ICPS architecture, as logically positioned in Fig. 3. The function of this layer is to present system status data, processed and analyzed by the cloud monitoring cluster, to human operators in a graphical or textual format, and to receive control commands from them. Therefore, the reliability and correctness of the software at this layer are directly related to the effectiveness of operator decisions and the ultimate security of the entire system.

In this section, we aim to analyze an extremely covert and significantly destructive B3 (Verification-Evasive) SFB. The uniqueness of such an SFB lies in the fact that its trigger condition is not caused by a single, local input, but is induced by the combined effect of the entire distributed system under a specific anomalous state. Such flaws are almost impossible to discover in traditional unit or integration testing because test environments can hardly replicate complex fault scenarios that span multiple layers.

We continue the failure chain from the previous section: the cloud monitoring cluster has fallen into a state of crashing and data corruption due to a protocol-parsing SFB. At this point, an operator attempts to query the system status or perform diagnostic operations using the application software on the industrial monitoring station. The station's software sends a request to the backend API, which is either non-operational or returning anomalous data.

Assume that a verification-evasive SFB exists within the monitoring station's application. This flaw is located in the module responsible for processing the response data from the cloud API. Under normal circumstances, this module would perform strict validation on the format and values of the returned data. However, this SFB causes a critical validation logic in the program to be completely bypassed when it receives a combination of a very rare negative value in a specific field (appearing due to the upstream data corruption) and a "service unavailable" status in another field.

This SFB, triggered in the final link, aggregates the consequences of failures from all previous layers and impacts the decision-maker in the most direct way, leading to the ultimate security issues:

- *Core threat-total subversion of decision integrity*: Because the critical validation logic is bypassed, the application does not crash but continues to execute. However, it renders the interface based on a contaminated and unvalidated data structure. This can lead to two extremely dangerous outputs:
  - False negative: The monitoring interface displays that the entire system is "all normal", with green lights everywhere. This misleads the operator into believing that no underlying issues have occurred, thereby missing the optimal window to take emergency measures (such as a manual shutdown) and allowing the physical-layer fault to continue to escalate.
  - False positive: The interface might, based on the erroneous data, mark a perfectly healthy device as the root cause of the failure on a topological map. This could induce the operator to perform incorrect interventions on the healthy device, such as rebooting or isolating it, thereby causing new, human-induced failures and exacerbating the chaotic situation.

  In both scenarios, the operator's decision integrity is fundamentally undermined.
- *Secondary threat: loss of HMI availability*: In some cases, although the validation is bypassed, the subsequent rendering logic cannot handle this unexpected internal data state, ultimately causing the application to crash, freeze, or become unresponsive. This results in the operator completely losing any visibility or control over the system, tantamount to the failure of the system's "last line of defense".

This case reveals one of the most severe security challenges in distributed systems: a fault can propagate unimpeded along the system architecture, from the lowest-level physical devices to the final link of human interaction. This final SFB, whose trigger conditions span the physical layer (device anomaly), the fog computing layer (network interruption), the cloud computing layer (data corruption), and the application layer (API request), demonstrates the cross-layer combined-effect characteristic of SFBs. It not only shatters the operator's trust in the entire monitoring system but also, due to its extreme difficulty of reproduction, makes the diagnosis and repair of the problem exceptionally challenging, posing a final and most severe blow to the long-term trustworthiness of the system.

5.6 Case study validation: mapping analysis based on the 2015 Ukraine power grid attack

To validate the practical significance of the Cascading Failure Model proposed in Sections 5.1 through 5.5, we conducted an in-depth analysis of the 2015 Ukraine power grid cyberattack. As a paradigmatic example, this incident vividly demonstrates how failures propagate across the various layers of a distributed Industrial Cyber-Physical System (ICPS). The attack vectors observed in this event align closely with the architectural analysis presented in this paper.

- **Mapping to the application layer (Initial intrusion and loss of control):** Similar to the "Application Layer Failure" scenario in our model, attackers first compromised operator workstations (the Application Layer) using BlackEnergy malware. This enabled them to hijack the Human-Machine Interface (HMI) software, thereby gaining unauthorized control over the system's decision-making interface. This breach severely undermined the operators' visibility and control, establishing the foundation for subsequent destructive actions.
- **Mapping to the fog/edge computing layer (Firmware-level denial of service):** A critical component of the attack involved the targeted manipulation of edge device firmware. Attackers targeted the serial-to-Ethernet gateways (specifically Moxa devices) that connected the control centers to the substations. By uploading malicious firmware or destructive data, the attackers rendered these devices inoperable ("bricking"). This action directly validates the **Fog Layer Infrastructure Fault** discussed in Section 5.3. In this context, destruction at the firmware level severed the communication link between the cyber and physical domains, disabling remote recovery mechanisms and forcing the system into an unavailable state.
- **Mapping to the physical layer (Cascading consequences):** The accumulation of failures in the aforementioned layers ultimately resulted in catastrophic consequences at the Physical Layer. With the Application Layer's HMI hijacked and the Edge Layer's communication gateways paralyzed by firmware attacks, attackers successfully opened circuit breakers at 30 substations, causing power outages for approximately 225,000 customers. This fact strongly validates our model's assertion: semantic logic flaws or exploited firmware vulnerabilities at the digital level inevitably cascade downward, ultimately leading to substantial physical process disruption.

In summary, this real-world case confirms that the "cascading failure" path is a genuine threat vector in distributed ICPS. It underscores that single-layer defenses are insufficient and necessitates the adoption of the system-wide security analysis approach proposed in this paper to mitigate such complex cross-layer risks.

5.7 Conclusion fo security issues

In summary, this multi-layer analysis has provided a concrete answer to RQ2, demonstrating how a seemingly localized SFB—specific to the categories defined in our taxonomy—can trigger a catastrophic, system-wide failure. By compromising integrity, availability, and confidentiality across every level of the distributed architecture, we have shown that the threat of SFBs is not merely theoretical but operationally critical.

The devastating potential of these cascading failures naturally

raises a pivotal question: Are current defensive mechanisms capable of handling such complex, cross-layer propagation? Consequently, the following section shifts our focus from problem analysis to solution evaluation. We will critically examine mainstream approaches for SFB detection and repair, evaluating their adequacy against the severe and multifaceted challenges identified in this scenario.

6 Analysis of mainstream security solutions: from traditional ICPS to distributed ICPS

Based on the preceding discussion of SFBs and their security impact in distributed ICPS, in this section, we undertake a critical examination of existing solutions. However, the objective of this paper is not to merely catalogue these solutions, but to conduct a deep analysis of the core principles and inherent assumptions that underpin them. To this end, we deconstruct the mainstream firmware bug detection and repair approaches into several key technical paradigms, classified by their fundamental principle. By scrutinizing the working mechanism, key techniques (including their mathematical foundations where applicable), and the intrinsic strengths and limitations of each paradigm, we aim to answer RQ3: *What are the mainstream detection and repair methods for SFBs, and what limitations exist in distributed contexts*? This analysis will systematically examine the paradigms for both detection and repair, laying the necessary groundwork for understanding and ultimately addressing the unique security challenges posed by distributed ICPS.

6.1 The evolution of SFB detection: from static rules to large language models

To understand the current landscape of SFB detection, it is essential to trace its technological trajectory. The field has evolved through three distinct stages, each emerging as a response to the limitations of its predecessor. We organize these methods chronologically and methodologically into: Early formal & knowledge-driven based, behavior-based anomaly, and learning-based detection approaches.

6.1.1 Stage I: formal & knowledge-driven analysis

Among the most classic approaches to SFB detection is formal & knowledge-driven based detection. As depicted in Table 3, this method primarily operates through pattern matching utilizing expert-defined prior knowledge. It can be summarized as verifying prior knowledge through experimental matching. This approach relies on a knowledge base built by human security experts from past experiences, incident analyses, and vulnerability research–a series of informal "experiments". It then employs a verification-like matching process to search for features within the target code that align with known defect patterns. This methodology transforms the open-ended problem of security auditing into a verification task based on established axioms.

This stage primarily encompasses two representative techniques: manual code review and pattern-based static analysis. Manual code review is the most direct embodiment of this paradigm, leveraging the tested knowledge and intuition within an expert's mind to scrutinize code logic.

To scale this expertise, researchers have developed automated tools. In the domain of binary static analysis, for instance, BinGo [37] employs selective inlining, architecture-neutral filtering, and length-variant partial traces, combined with TF-IDF and cosine similarity, to automatically detect semantically similar binary functions across architectures (e.g., x86/ARM) and OSs (e.g., Linux/Windows), performing efficiently on datasets like coreutils and BusyBox. DiscovRE [38] uses automated numeric and structural feature extraction from control flow graphs to perform efficient cross-architecture similarity search for known bugs in large binary codebases without manual signature engineering.

From the other standpoint, the essence of these methods is a sophisticated form of "template matching", where the matching process itself can be seen as a form of experimental verification. At the source code level, the "experiment" involves checking whether the code adheres to specific secure coding standards. In the more complex realm of binary analysis, this experiment evolves into a graph-theory matching problem. Analysis tools first convert a target function into a control flow graph (CFG), while a known vulnerability pattern is abstracted into a "vulnerability CFG template." The detection "experiment" then consists of verifying the existence of a subgraph within the target's CFG that is isomorphic to the vulnerability template. This process can be abstractly modeled as follows:

$$\text{Result} = \text{Match}(\textit{TargetCode_Representation}, \textit{Predefined_Ruleset}), \quad (1)$$

where $TargetCode_Representation$ is the representation of the code, and $Predefined_Ruleset$ is the knowledge base derived from past experience.

The primary advantage of this paradigm lies in its precision and high fidelity. Because the "experiment" targets a well-defined, known defect, a positive result (i.e., a successful match) typically yields a high accuracy rate with few false positives. The logic behind a detection is clear, making it easy for security personnel to understand, reproduce, and remediate—a critical feature in scenarios requiring rapid incident response.

However, the methods' fundamental limitations in this stage are also a direct consequence of its core principle. First, it is incapable of discovering novel SFBs that are not defined in its knowledge base. Its capability is bounded by the scope of past "experiments", rendering it ineffective against "zero-day" [39] semantic defects that exploit entirely new techniques. Second, the cost of constructing and maintaining the knowledge base is prohibitive. The process of translating unstructured expert knowledge and vulnerability analysis results into precise, robust, and machine-executable "experimental" rules is both challenging and time-consuming. Finally, this approach exhibits poor scalability when faced with new hardware architectures or programming paradigms, as each new context may require designing an entirely new set of "experimental protocols" (i.e., rulesets). These inherent constraints make this stage increasingly inadequate for confronting the complexity of modern ICPS firmware.

Table 3 Comparative analysis of SFB detection approaches

Detection approach	Strength(s)	Limitation(s)	Repre-Ref/Year	Implementation details (Core Idea)
Formal & Knowledge-Driven	1. High precision and fidelity for known bugs. 2. Reproducible results.	1. Cannot detect 0-day bugs. 2. High cost for creating and maintaining the knowledge base. 3. Poor scalability.	Manual Code Review (N/A)	Extracts length-variant partial traces from binaries and then measures their similarity using Term Frequency-Inverse Document Frequency (TF-IDF) and cosine similarity.
			Bingo [37], 2016	Also codifies expert knowledge into vulnerability signatures for automated pattern-based static analysis.
			discovRE [38], 2017	Enables signature-free bug detection across architectures by automatically extracting numeric and structural features from control flow graphs.
Behavior-based Anomaly Detection	1. Detect unknown (0-day) bugs. 2. Cross-platform applicability.	1. Low coverage. 2. Cannot define baseline. 3. High sensitivity to the test environment.	Blanket [39], 2014	Proposes blanket execution, addressing binary function similarity challenges through dynamic analysis.
			CaCompare [38], 2017	Detects cross-architecture binary clones via semantic emulation.
			MultiFuzz [41], 2024	Fuzzes firmware with multi-stream inputs, solving peripheral interaction challenges and finding new bugs.
Learning-based Semantic Analysis	1. Automatic. 2. Learns complex semantic patterns.	1. Heavy reliance on high-quality labeled data. 2. Representation gap. 3. Hard to seek system level emergent SFBs.	Genius [44], 2016 (Machine Learning-based)	Extracts statistical features from Control Flow Graphs (CFGs) and uses spectral clustering to find homologous (similar) functions.
			Gemini [47], 2017 (Graph Neural Network (GNN)-based)	Uses a graph embedding network (Structure2vec) on Attributed Control Flow Graphs (ACFGs) to learn function embeddings.
			α-Diff [54], 2018 (Convolutional Neural Network (CNN)-based))	Uses a CNN network on raw bytes with three semantic features for effective cross-version binary similarity detection.
			VulDeePecker [56], 2018 (Recurrent Neural Network (RNN)-based))	Defines a novel program representation "Code Gadget" to serve as input for a Bidirectional Long Short-Term Memory (BLSTM).
			OrderMatters [63] (Attention-based), 2020	Fuses Bidirectional Encoder Representations from Transformers (BERT) semantic information with ResNet [60] order-awareness to improve binary code similarity detection.
			VulLLM [66], 2024 (Large Language Model (LLM)-based)	A framework that improves code bug detection by fine-tuning Large Language Models with multi-task learning.

6.1.2 Stage II: behavior-based anomaly detection

Different from the Stage I methods, which rely on prior knowledge, behavior-based anomaly detection adopts an inductive, rather than experience. In Table 3, its core idea is to first establish a baseline of a program's "normal" behavior, subsequently identifying any dynamic execution that deviates from this baseline as a potential anomaly. In this stage, the most methods are not directly concerned with the structure of code; instead, they focus on the externally observable effects a program exhibits during its actual execution. This makes them naturally suited for discovering dynamic defects triggered only by complex state interactions or unexpected inputs. The technical implementation of this stage is dominated by two mainstreams: one is dynamic analysis that monitors and compares execution traces, as represented by the work of Blanket [39] and CaCompare [38]; the other one is fuzzing [40] based detection, a cornerstone technique in the behavior-based anomaly detection paradigm, designed to proactively induce anomalous behavior by automatically generating large volumes of semi-random input data to uncover potential bugs. For fuzzing related work, MultiFuzz [41] presents a multi-stream input approach for fuzzing firmware. By creating independent input streams for each hardware peripheral, it overcomes the inefficiencies of traditional single-file inputs, enabling more targeted mutation and extension strategies to significantly improve code coverage and bug discovery.

Mechanistically, this stage aims to capture and quantify a program's dynamic behavior. Execution trace comparison techniques typically employ program instrumentation to implant probes at critical paths (e.g., function calls, system calls), thereby recording a sequence of events at runtime to form a "behavioral fingerprint" or trace. The detection process then involves comparing a test trace against a pre-established "normal" baseline. This comparison can be formalized mathematically: if an execution trace T is treated as a set of discrete events, the similarity between two traces, T_A and T_B, can be calculated using metrics like the Jaccard similarity coefficient:

$$\text{Similarity}(T_A, T_B) = \frac{|T_A \cap T_B|}{|T_A \cup T_B|}. \quad (2)$$

An anomaly is flagged when the similarity between a test trace and a baseline trace falls below a certain threshold. Fuzzing, on the other hand, adopts a more aggressive strategy. It utilizes a high-throughput input mutation engine to generate a vast number of test cases while continuously monitoring the program for explicit signs of anomaly, such as a crash, an assertion failure, or undefined behavior caught by memory safety sanitizers (e.g., AddressSanitizer [42]). Modern coverage-guided fuzzers further enhance this process by using lightweight instrumentation to track code coverage, prioritizing the mutation of inputs that explore new code paths, thereby significantly increasing the efficiency of bypassing complex logic checks to trigger deep-seated bugs.

The primary advantage of this stage lies in its exceptional ability to discover unknown bugs. By not relying on any prior vulnerability signatures, it can uncover novel, "zero-day" SFBs, particularly logic flaws that are tightly coupled with specific runtime states. Furthermore, its focus on the behavior of an executable rather than its implementation details grants it excellent cross-platform and cross-architecture applicability. The astounding effectiveness of fuzzing in finding memory corruption vulnerabilities has established it as an important technology in firmware security testing.

However, the intrinsic constraints lie beneath this stage's seemingly powerful capabilities. Its disadvantage is the challenge of code and state-space coverage. For any non-trivial firmware program, the number of potential execution paths undergoes a combinatorial growth, a problem known as "path explosion". Within any finite testing time, it is computationally infeasible for either dynamic analysis or fuzzing to traverse all paths, let alone cover all possible state combinations. The fuzzing related tools are often blocked from reaching core logic by complex checksums or "magic number" checks. Secondly, establishing a "normal" behavioral baseline is an intractable problem in itself. For trace comparison methods, a seemingly normal baseline may already contain latent, untriggered flaws. Concurrently, minor environmental perturbations (e.g., OS scheduling) can cause significant trace deviations, leading to a high rate of false positives. Conversely, a critical logic vulnerability that does not cause a crash can be completely missed, resulting in false negatives. Lastly, this stage is highly sensitive to the test environment; a finding from a laboratory setting may not be reproducible in a production environment, posing significant challenges for bug validation and remediation.

■ 6.1.3 Stage III: learning-based semantic information analysis

Even with their advantages, knowledge-driven code analysis methods are constrained by the boundaries of expert knowledge, while behavior-based approaches encounter challenges due to the combinatorial explosion of state spaces. To address these limitations, learning-based semantic analysis has emerged, aiming to enable machines to autonomously learn deep semantic logic information from large-scale code datasets, thereby understanding complex concepts such as vulnerable code fragment. The essence of the learning-based semantic analysis paradigm lies in the transformation of discrete, structured code into a continuous, high–dimensional mathematical representation that machines can process, a process known as "code representation" [42–43]. The evolution process of this field is, in essence, the technology evolution of an evolving code representation technology. In this section, we deviate from a strict chronological order and instead dissect the paradigm according to several dominant representational strategies: viewing code as a collection of *statistical features*, as a *topological graph*, as a *tensor*, as a *textual sequence*, and finally, as an *all-encompassing semantic space* defined by large-scale pre-trained models. This analysis will ultimately reveal their shared limitations when confronted with the security challenges of distributed ICPS.

■ 6.1.3.1 Origin of the idea: machine learning based on statistical features

Before deep learning based approach became mainstream, researchers employed sophisticated feature engineering to quantify the structure and attributes of code into numerical features, which were then analyzed using classical machine learning algorithms. The underlying idea is that code semantic information can be approximated through measurable statistical properties. The theoretical framework can be summarized as mapping a code unit c (e.g., a function) to a feature vector ($v \in \mathbb{R}^n$) through a manually designed feature extraction function Φ (i.e., ($v=\Phi(c)$)). These vectors are subsequently used as input for traditional machine learning algorithms. In the context of SFB detection, unsupervised clustering is particularly common. For example, the spectral clustering algorithm does not directly operate on vectors but works on a similarity graph constructed from these vectors. By computing the Laplacian matrix (L=D-A) and utilizing the feature vectors corresponding to its smallest vectors, the algorithm can identify an optimal partition of the graph, thereby grouping semantically similar functions together.

The pioneering work in this strategy was Genius [44], which precisely implemented the aforementioned framework by defining (Φ) to extract topological and instruction set features from the control flow graph (CFG) of a function and applying spectral clustering to discover homologous SFBs. Following closely, BinHunt [45] and its improved version iBinHunt [46], though reliant on symbolic execution, provided significant inspiration for subsequent feature extraction by formalizing the comparison of basic block semantic information.

■ 6.1.3.2 Embracing topology: treating code as graph structures (GNN)

To overcome the limitations of manual feature engineering, the research focus has shifted to leveraging the natural graph structures of code (i.e., abstract syntax tree (AST), control flow graph (CFG), logic semantic flow graph (LSFG), program dependence graph (PDG), and code property graph (CPG)) and employing graph neural networks (GNN) to automatically learn their deep structured representations. This strategy represents code as a graph, defined as follows:

$$G = (V, E), \tag{3}$$

where V denotes nodes (e.g., functions, basic blocks), and E denotes edges (e.g., control flow, data flow). GNN learns node embeddings through a "message passing" mechanism. In each layer l, a node u aggregates messages $m_{v\to u}$ from its neighbors $v \in N(u)$ and combines them with its own state $h_u^{(l)}$ to update its new hidden state $h_u^{(l+1)}$. This process can be formalized as

$$h_u^{(l+1)} = \mathrm{UPDATE}^{(l)}\left(h_u^{(l)}, \mathrm{AGGREGATE}^{(l)}\left(\{m_{v\to u}^{(l+1)} \mid v \in N(u)\}\right)\right), \tag{4}$$

where $N(u)$ denotes the set of neighbor nodes of u, and $m_{v\to u}^{(l+1)}$ represents the message vector passed from neighbor v to u.

Among numerous GNNs, graph convolutional networks (GCNs) process graph structural information through neighborhood normalization, providing an efficient aggregation mechanism that enhances the model's capability to represent code semantics. Specifically, GCN assumes that node features should be updated through weighted averaging of their neighborhoods, which is suitable for the sparsity and hierarchy of code graphs, enabling better capture of control flow and data flow dependencies, avoiding gradient issues in standard GNNs on uneven graphs (such as CFG), thereby improving the precision of SFB detection. Its layer-wise propagation rule is as follows:

$$H^{(l+1)} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right), \tag{5}$$

where $H^{(l)}$ is the feature matrix of all nodes in layer l, and the term $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$ serves the role of normalized aggregation of neighborhood features.

Innovations in this field primarily lie in the selection of graph representation G and the design of specific GNN architectures (i.e., the UPDATE and AGGREGATE functions). Gemini [47] is the foundational work in this direction, employing attributed control flow graph (ACFG) and a specially designed graph embedding network Structure2vec [48] to detect similarity in binary functions. Devign [49] became a benchmark for source code SFB detection, constructing a rich code property graph and utilizing gated graph neural network (GGNN) for learning. Other works such as VulSeeker [50], BinSeeker [51], and TAILOR [53], apply GNN to various scenarios like binary SFB search and code differencing, fully demonstrating the considerable capabilities of graph models.

■ 6.1.3.3 A different path: treating code as "images" (CNN)

Unlike methods focusing on topological structures, another technical strategy draws inspiration from computer vision, treating code (particularly its binary form, i.e., a sequence of "0" and "1") as an "image". This strategy maps binary code directly into a tensor and utilizes convolutional neural networks (CNNs) to extract local feature information. It relies on a mapping function (M:C→I), which converts code (C) into a one-dimensional or two-dimensional matrix (I) (i.e., an "image"), where pixel values may represent byte values or instruction categories. CNNs perform convolution operations on this matrix using a convolutional kernel K:

$$((I * K)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n)), \tag{6}$$

where I represents the input matrix (the "image" derived from the binary code), K is the convolutional kernel (a small matrix of weights), and (i, j) denotes the position in the output feature map. The double summation over m and n iterates through the rows and columns of the kernel, respectively, computing a weighted sum of the input values centered around (i, j). This operation effectively detects local patterns, such as edges or textures in the "image", by sliding the kernel across the entire input matrix and producing a feature map that highlights these extracted features. In CNNs applied to binary code analysis, this helps identify recurring instruction patterns or structural motifs without relying on explicit syntactic parsing.

This process is essentially a local pattern-matching task, where the network learns, through training, convolutional kernels that produce strong activations for specific byte sequences or instruction arrangements (e.g., function prologues, parameter settings of dangerous APIs). α-Diff [54] is a typical application of this strategy, treating the instruction embedding sequence within basic blocks as a one-dimensional "image" and designing a siamese CNN architecture to learn and compare their semantic information. VulCNN [55] is a typical application of this strategy, treating the source code as a two-dimensional "image" and designing a CNN architecture to learn and detect vulnerabilities based on semantic features. The work of Russell [52] demonstrates that this idea is equally applicable to source code, applying a one-dimensional CNN to source code text to effectively detect vulnerabilities caused by local problematic patterns. These works prove that code also exhibits meaningful patterns in local spaces that can be leveraged for learning.

■ 6.1.3.4 Back to the source: treating code as sequences (RNN/LSTM)

Treating code as a sequence is one of the most natural and earliest attempts to apply deep learning to this domain. Drawing on mature techniques from natural language processing (NLP), code is "flattened" into a one-dimensional sequence of tokens $(x_1, x_2, \ldots, x_T)$, and recurrent neural networks (RNNs) or their variants are used to capture dependencies within the sequence.

A standard RNN processes the sequence through the recursive relation:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h), \tag{7}$$

where h_t is the hidden state at time step t, W_{hh} and W_{xh} are weight matrices, b_h is the bias, and f is a non-linear activation function. However, due to the vanishing gradient problem, RNNs struggle to learn long-range dependencies.

To address this, long short-term memory networks (LSTMs) are widely adopted. LSTMs introduce an additional cell state c_t and three sophisticated gating mechanisms: the input gate i_t, forget gate f_t, and output gate o_t. These gates allow the model to selectively forget, input, and output information, effectively learning long-range associations in code, such as those between variable declarations and their subsequent uses. The core state transition formulas are

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_c \cdot [h_{t-1}, x_t] + b_c), \quad (8)$$

where $\odot$ denotes element-wise multiplication, W_c is the weight matrix for the cell state, b_c is the bias, and $[h_{t-1}, x_t]$ represents the concatenation of the previous hidden state and current input.

$$h_t = o_t \odot \tanh(c_t), \quad (9)$$

where $\odot$ denotes the element-wise Hadamard product, f_t, i_t, o_t represent the activations of the forget, input, and output gates respectively, and c_t is the cell state vector.

The first equation updates the cell state c_t by combining the previous cell state c_{t-1}, modulated by the forget gate f_t, with new information from the input gate i_t and a candidate cell state computed via tanh. The second equation computes the hidden state h_t by applying the output gate o_t to the transformed cell state $\tanh(c_t)$. This structure enables LSTMs to maintain and update long-term dependencies effectively.

Among the tools in this strategy, VulDeePecker [56] is a representative work that uses bidirectional LSTM (BLSTM) [57] to detect vulnerabilities in code snippets, laying the foundation for subsequent research. Later, works like SySeVR [58] built upon this approach. The study OrderMatters [59] deeply explores the importance of sequence order in code representation, demonstrating that well-designed sequence models can, in certain scenarios, perform comparably to graph-based models. Additionally, works like DeepJIT [61] successfully applied sequence models to more practical software engineering tasks, such as predicting whether a code commit will introduce defects.

■ 6.1.3.5 Frontier of innovation: large-scale pre-trained language models (LLM)

For now, the current research frontier is undoubtedly dominated by large-scale pre-trained models based on the Transformer [62] architecture. These models, through "pre-training and fine-tuning" on vast amounts of text and code data, learn general, context-rich semantic information representations and exhibit remarkable reasoning and generation capabilities. The core innovation of the Transformer architecture is its self-attention mechanism, which abandons the recurrent structure of RNNs in favor of parallelized, direct pairwise token interactions. For an input sequence, this mechanism computes three matrices: query (Q), key (K), and value (V), with the output being a weighted sum of the values, where the weights are determined by the similarity between queries and keys:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (10)$$

where Q, K, V are the query, key, and value matrices, and d_k represents the dimension of the key vectors, used as a scaling factor to prevent vanishing gradients.

The self-attention mechanism allows the model to incorporate the importance of all other tokens in the sequence when representing any given token, thereby capturing highly complex long-range contextual relationships. The term $\frac{QK^T}{\sqrt{d_k}}$ computes the similarity scores between the query and key vectors, scaled by the square root of the key dimension d_k to stabilize gradients. The softmax function normalizes these scores into weights, which are then used to compute a weighted sum of the value vectors V. Large language models (LLMs) achieve powerful zero-shot or few-shot learning capabilities by pre-training on massive datasets with objectives like masked language modeling (MLM), which forces large Transformer models to internalize deep syntactic and semantic patterns of code.

From CodeBERT [63], which served as a bridge, and its evolved version GraphCodeBERT [64], which incorporates graph information, to the widely recognized code generation model GitHub Copilot (OpenAI Codex), and more advanced multimodal models like UniXcoder [65], LLMs are transforming bug detection from a classification problem into an interactive task that can directly perform auditing and suggest fixes through "prompt engineering". Numerous evaluations of general-purpose models like GPT-4 also demonstrate their potential in identifying simple vulnerabilities, marking a profound paradigm shift in this field. For the LLM-based bug detection work, VulLLM [66] proposes a framework that improves code vulnerability detection by fine-tuning Large Language Models with multi-task learning. It compels the model to not only detect vulnerabilities but also localize them using patches and explain their root causes, thereby learning deeper semantic features for enhanced generalization and robustness. Twisted-CodeBERT-GCN [67] is a semantic learning framework that integrates Logic Semantic Flow Graphs (LSFGs) with LoRA-fine-tuned CodeBERT representations to capture multi-modal semantics, enabling the simultaneous detection of single-device bugs (SDBs) and cross-device bugs (CDBs) in deployed IoT systems. LLMxCPG integrates code property graphs with large language models for vulnerability detection [68].

■ 6.1.3.6 Common shortcomings and the representation gap of learning-based strategy in distributed ICPS

Although the aforementioned methods exhibit a diversity of choices in code feature representation and have each achieved significant success in bug detection, when we turn our attention to distributed ICPS, their shortcomings become exposed:

- **Dependence on high-quality labeled data.** Regardless of the theoretical basis of the models, they all heavily rely on high-quality labeled data, yet public datasets describing "distributed system collaboration failures" are virtually nonexistent. This shifts the supervised learning paradigm in this specific domain from "data scarcity" to a "data desert."
- **Gap in representation capabilities.** The more fundamental issue lies in the gap in representation capabilities. Reviewing their technological evolution, all models' design paradigms are limited to analyzing a closed, monolithic program context. They excel at understanding the logic within a function, a file, or a project, but their model structures themselves cannot effectively model the interaction semantics between heterogeneous components across networks, protocols, and vendors. Even for powerful large language models (LLMs), their training data and architectures primarily focus on understanding the internal logic of a single codebase, rather than the temporal dependencies and protocol interactions

between multiple heterogeneous systems.

- **System-level emergent SFBs.** The root cause of a distributed SFB may not be a defect in the code of any single component, but rather the unintended system-level behavior triggered jointly by message M sent by component A at a specific time, component B's internal state S, under a network delay of T. This kind of vulnerability semantics that is "system-level and emergent," arising from the interplay of time, state, and communication, represents a "representation blind spot" that all existing learning paradigms struggle to reach, and it is also the key challenge that subsequent research in this paper needs to address.

6.1.4 Performance analysis and benchmarking of SFB detection

While the previous sections provide a qualitative taxonomy, Table 4 summarizes the quantitative metrics extracted from representative literature to illustrate the performance evolution of SFB detection methods.

Performance trends: As shown in Table 4, there is a clear trajectory in balancing accuracy and efficiency. Early knowledge-driven methods like discovRE achieved high precision but suffered from significant computational overhead during graph matching. The introduction of learning-based paradigms marked a turning point; for instance, Genius reduced search times to the sub-second level by utilizing spectral clustering. Subsequent GNN-based models like Gemini further optimized this, reducing training time from weeks to minutes while maintaining high AUC scores. More recent LLM-based approaches demonstrate superior semantic understanding, albeit with higher inference costs suitable for offline analysis rather than real-time ICPS loops.

Comparison limitations: It is crucial to note that a direct "apples-to-apples" comparison across all methods is inherently challenging due to the *heterogeneity of experimental setups*.

- **Datasets:** Methods operate on different inputs, ranging from source code slices (e.g., SARD) to binary control flow graphs extracted from firmware.
- **Metrics:** Search-based tools often prioritize Recall and Rank, whereas classification tools prioritize F1-score and False Positive Rate.
- **Environment:** The transition from standard architectures to distributed ICPS testbeds introduces variables (e.g., network latency) not present in static dataset benchmarks.

Therefore, the data in Table 4 should be interpreted as indicative of technological progress rather than an absolute ranking.

6.2 Automatic SFB repair (ASR): four core technical paradigms

Following SFB detection, automatic SFB repair (ASR) serves as the second critical line of defense against software defects. Its goal is to automatically generate correct and acceptable patches to reduce the vulnerability exposure window and lower the high costs of manual repair. However, in this section, we systematically argue that, despite the evolution of ASR through multiple paradigms, its core assumptions and technical approaches reveal limitations that are not only similar to those in detection techniques but even more important when addressing system-level functional behavior defects in distributed ICPS.

6.2.1 Paradigm 1: search-based heuristic SFB repair

In Table 5, the earliest ASR paradigm formalizes patch generation as a search problem within a vast space of code modifications. Grounded in search-based software engineering (SBSE), this paradigm is epitomized by genetic programming (GP) [69]. GP treats a defective program as an "individual" in a population, generating new program versions (candidate patches) by simulating evolutionary operators such as crossover and mutation. The core principle of this process is the fitness function, which quantifies the quality of each candidate patch. A typical fitness function, $Fitness(P')$, is defined as:

$$\text{Fitness}(P') = w_{\text{pos}} \cdot |\mathcal{T}_{\text{pos}}(P')| - w_{\text{neg}} \cdot |\mathcal{T}_{\text{neg}}(P')|, \tag{11}$$

where $\mathcal{T}_{\text{pos}}$ is the set of passing test cases that validate existing functionality, while $\mathcal{T}_{\text{neg}}$ is the set of failing test cases that expose the original defect. The terms w_{pos} and w_{neg} are weights. An ideal patch must pass all failing test cases while not regressing on any passing ones. Seminal works such as RSRepair, and AE [70–71] all adhere to this framework. However, the fundamental limitations of this paradigm are the combinatorial explosion of the search space and its strong dependency on high-quality test suites. More critically, the

Table 4 Quantitative performance analysis of representative SFB detection methods

Model / Year	Dataset	Key Metrics
Genius [44], 2016	OpenSSL, BusyBox, Coreutils (Cross-Arch)	AUC: ~0.90 Time: ~0.1s/func
Gemini [47], 2017	OpenSSL, Firmware Images (Cross-Arch)	AUC: 0.971 Train: 30m/epoch
α-Diff [54], 2018	Coreutils, Diffutils (Cross-Version)	Recall@1: 0.78 Acc: ~0.85
VulDeePecker [56], 2018	NVD, SARD (C/C++ Source)	F1: 92.9% FPR: 3.4%
VulLLM [66], 2024	Real-world CVEs (Multi-task)	Acc: 94.5% F1: 93.8%
Twisted-CodeBERT-GCN [63], 2025	NVD, SARD (C/C++ Source)	Acc: 89.6% (Zero-shot)

Table 5 Comparative analysis of SFB repair paradigms

Repair paradigm	Strength(s)	Limitation(s)	Repre-Ref/Year	Implementation details (Core Idea)
Search-based Heuristic SFB Repair	1. Fully automated. 2. Does not require formal specifications or models.	1. Search Space Explosion: Scalability is severely limited. 2. Test-Suite Overfitting: Often generates plausible but incorrect patches. 3. Low patch correctness rate.	GP [69], 2009	Uses genetic programming to evolve program code, treating the test suite as the fitness function to guide the search for a valid patch.
			RSRepair [70], 2014	Employs a random search strategy, which is simpler but less guided than genetic programming, to find a working patch.
Constraint-based SFB Repair	1. Higher patch correctness compared to heuristic search. 2. Patches are synthesized to satisfy specific semantic constraints.	1. Scalability Issues: Satisfiability Modulo Theories (SMT) and Boolean Satisfiability Problem (SAT) solvers can be very slow on large, complex programs. 2. Still dependent on the quality of the test suite to derive constraints. 3. Limited to specific types of bugs.	SemFix [72], 2013	Uses symbolic execution to extract path constraints from failing tests and an SMT solver to synthesize repairs that satisfy these constraints.
			Nopol [74], 2016	Specializes in fixing conditional bugs by using an SMT solver to synthesize new or modified `if` conditions.
Learning-based SFB Repair	1. Learns repair patterns from bug-fix datasets. 2. Can fix a wider variety of bugs than constraint-based methods.	1. Data-Dependent: Performance is tied to the training data; struggles with novel bugs. 2. Lacks formal correctness guarantees. 3. Generated code can be non-idiomatic.	DeepFix [77], 2020	An end-to-end deep learning system that iteratively fixes common C compilation errors using a Seq2seq model.
			CoCoNuT [78], 2020	A context-aware Neural Machine Translation (NMT) ensemble for effective, multi-language program repair.
LLM-based Generative Repair	1. Leverages vast pre-trained knowledge. 2. Excellent zero-shot/few-shot repair capabilities. 3. Can understand natural language bug descriptions.	1. Hallucinations: High risk of generating plausible but incorrect or unsafe code. 2. Black-Box Nature: Difficult to verify or explain the reasoning process. 3. High computational cost for inference.	InferFix [82], 2023	An LLM-based program repair system guided by a static analyzer and retrieval-augmented prompts.
			RepairLLaMA [83], 2025	A LLM fine-tuned on code that can generate patches from context or instructions alone (in-context learning).

generated patches often overfit to the test cases, resulting in a large number of "plausible but incorrect" fixes.

6.2.2 Paradigm 2: constraint-based SFB repair

As shown in Table 5, to overcome the blindness of search, the second paradigm introduced program semantics to guide the repair process from the perspective of "why" the error occurs. This approach is centered on formal methods like symbolic execution. When symbolic execution detects an error (e.g., a failed assertion) along a specific path, it generates a path constraint, ψ, that precisely describes the condition for the error to occur. The program repair task is thereby transformed into a logical constraint-solving problem: synthesize a patch p at a specific code location such that the new path constraint ψ' satisfies the logical implication $\psi' \implies \neg\psi$. Patch synthesis is typically performed within a predefined space of syntactic templates, and a constraint solver is used to find a concrete implementation that fulfills the logical condition.

Works represented by SemFix, Angelix, and Nopol [72–74] have successfully applied this theory to repair specific defect classes, such as errors in conditional expressions. By leveraging program semantics, these methods achieve a significantly higher rate of correct patches compared to search-based approaches. Yet, their bottlenecks are equally pronounced: the inherent path explosion problem of symbolic execution limits their scalability to large, complex programs. Furthermore, the types of defects they can handle are limited to those that can be clearly expressed by logical constraints, leaving them powerless against errors in complex algorithms or data structures.

6.2.3 Paradigm 3: learning-based SFB repair

Following breakthroughs in Natural Language Processing (NLP), researchers began to frame program repair as a "translation" problem: translating from a "buggy program language" to a "correct program language". This paradigm predominantly employs the Sequence-to-Sequence (Seq2Seq) architecture, which consists of an Encoder and a Decoder, connected via an Attention Mechanism. The Encoder (e.g., an LSTM or Transformer) reads the defective code sequence and compresses it into a semantics-rich context vector. The Decoder then uses this vector to generate the fixed code sequence, token by token. Complementary work has explored LLM-based repair of introductory Python assignments [75] and LLM-based assessment of patch correctness [76].

The lifeblood of this approach is massive datasets of $(buggy_code, fixed_code)$ pairs, typically mined from the commit histories of large-scale open-source repositories like GitHub. The model learns repair patterns by generalizing from millions of human-written fixes. DeepFix, CoCoNuT, and SequenceR [77–79] are exemplary works of this paradigm. Despite achieving remarkable

success in fixing common, idiomatic bugs, their limitations are stark: model performance is heavily dependent on the scale and quality of training data, failing to generalize to novel bug types not seen during training; they frequently generate patches that are syntactically correct but semantically flawed; and their scope of repair is often local, with limited capability for fixing complex bugs that require global context.

■ 6.2.4 Paradigm 4: LLM-based generative repair

The current frontier leverages the vast pre-trained knowledge and instruction-following capabilities of Large Language Models (LLMs), transforming program repair from a pure synthesis task into an interactive reasoning and generation task. Unlike neural repair, which relies on supervised fine-tuning on paired data, LLMs utilize their zero-shot or few-shot capabilities to perform repairs directly via well-crafted prompts. A prompt can include the defective code, compiler errors, test failure logs, and even natural language descriptions of the repair requirements. The model, in turn, directly generates repair suggestions, code patches, and even explanations for the fix.

Research on models like OpenAI Codex, GitHub Copilot, and GPT-4 [80–81] demonstrates their surprising potential for fixing certain defects, exhibiting a degree of interactivity and explainability absent in previous paradigms. For instance, InferFix [82] leverages a static analyzer for precise bug detection and classification. It uses retrieval-augmented prompts, which provide relevant historical fixes, to guide the model in generating more accurate and context-aware patches for automated bug fixing. RepairLLaMA [83] creates a specialized program repair adapter by using efficient fine-tuning (LoRA) and tailored code representations. These representations incorporate crucial repair signals like fault localization and the original buggy code, enabling the model to effectively fix complex, multi-location bugs.

However, this paradigm is still in its nascent stages, with non-trivial inherent flaws: models are prone to "hallucination", potentially inventing non-existent APIs or introducing new, more subtle bugs; their outputs lack correctness guarantees, necessitating rigorous review by human experts; and their capabilities remain questionable for complex bugs that require deep, domain-specific knowledge, such as industrial protocols or OS kernels.

Table 6 summarizes the datasets and reported quantitative results of representative automated program repair models. Since these studies employ different benchmarks and evaluation settings, the results are intended to illustrate the evolution of repair capabilities rather than provide a direct performance ranking.

■ 6.2.5 The "Distributed Gap": why traditional ASR fails in ICPS

While the paradigms discussed above have shown promise in isolated software environments, applying them to Distributed ICPS reveals a fundamental "Distributed Gap." In a monolithic application, a bug fix is a local code transformation $P \rightarrow P'$. However, in a distributed ICPS, a repair action is a complex orchestration involving time, state, and topology [84]. The challenges are no longer just about generating correct syntax, but about preserving system-wide invariants during and after the repair process. We categorize these unique challenges into three critical dimensions: Cross-Node State Consistency, Protocol Semantic Alignment, and Real-Time Deployment Constraints.

■ 6.2.5.1 Challenge I: cross-node state consistency repair

In distributed ICPS, an SFB often leaves the system in a "corrupted state" distributed across multiple nodes. Merely patching the firmware code ($Code_{fix}$) on the faulty node is insufficient if the distributed state (S_{dist}) remains inconsistent [85].

- **The state residual problem:** Consider a "Leader-Follower" robotic cluster where a semantic bug in the Leader caused it to skip a synchronization signal. Even if we hot-patch the Leader's firmware to fix the logic, the Follower nodes may still be stuck in a "Waiting" state, expecting a signal that was never sent. A code repair without a state repair leads to a "Zombie System" where nodes run correct code but operate on stale or invalid context.
- **Repair atomicity:** In a distributed transaction, repairing one node while others continue to process data can introduce "Split-Brain" scenarios [86]. A repair strategy must ensure **Global Repair Atomicity**. Formally, if a repair R transitions the system from state S_t to S_{t+1}, it must satisfy the invariant I:

$$I(S_t) \wedge R \rightarrow I(S_{t+1}). \tag{12}$$

Current ASR tools are predominantly optimized for local scope, often limiting their capacity to verify I across network

Table 6 Quantitative performance analysis of representative automated program repair models

Model / Year	Dataset	Key Metrics
RSRepair [70], 2014	7 C Programs (24 Buggy Versions)	Efficiency: > GenProg (23/24 cases) NTCE: Significantly Lower
DeepFix [77], 2017	6,971 Student C Programs (93 Tasks)	Fixed: 1,881 (27%) Partial: 1,338 (19%)
CoCoNuT [78], 2020	Defects4J, QuixBugs, Codeflaws, ManyBugs	Total Fixed: 509 bugs Unique Fixes: 309
InferFix [82], 2023	InferredBugs (Java & C# Repositories)	Top-1 Acc (Java): 76.8% Top-1 Acc (C#): 65.6%
RepairLLaMA [83], 2025	Defects4J v2, HumanEval-Java, GitBug-Java	Fixed (D4J): 144 Fixed (HE-Java): 109

boundaries. Consequently, generating the necessary "State Migration Scripts" to reset or roll forward the states of remote peers remains a significant challenge for these approaches.

■ 6.2.5.2 Challenge II: protocol semantic repair

SFBs in ICPS frequently manifest as violations of protocol semantics rather than internal computation errors. For instance, a device might misunderstand the Quality of Service (QoS) level in an MQTT message, leading to data loss.

- **Implicit assumption mismatches:** Distributed-Native bugs (Type B) often arise from mismatched assumptions between heterogeneous vendors [87]. Vendor A's firmware might assume a "Keep-Alive" interval of 30s, while Vendor B assumes 60s. A repair here is not necessarily fixing a "wrong" line of code, but reconciling these semantic parameters.
- **The need for "Adapter" synthesis:** Sometimes, the source code of a third-party component is immutable (Black-box). In such cases, "repair" implies synthesizing an external **Protocol Adapter** or **Sanitizer** that sits between the network interface and the legacy firmware [86]. This adapter must intercept traffic and mutate packets to align with the correct protocol semantics on the fly. Current ASR paradigms, focused on source-code modification, are incapable of generating such architectural repairs.

■ 6.2.5.3 Challenge III: patch deployment under timing constraints

Perhaps the most severe constraint in ICPS is the requirement for high availability and real-time response. The "Stop-Patch-Restart" model used in IT systems is unacceptable in OT (Operational Technology) environments where a shutdown could mean production loss or safety hazards.

- **The real-time window problem:** An ICPS device operates in strict control loops (e.g., sense-compute-actuate every 10ms) [88]. A repair mechanism must perform **Hot-Patching** (dynamic software updating) within the idle time of a single control cycle [89]. If the patching process exceeds this window ($T_{patch} > T_{idle}$), it causes "Control Jitter," potentially destabilizing the physical process.
- **Safe state identification:** A semantic repair often requires the program to be in a "Quiescent State" (where no critical transactions are active) to be safely applied. In a continuous manufacturing line, identifying or forcing such a state without disrupting production is a complex control-theoretic problem, not just a software engineering one.

■ 6.2.6 The "Distributed Gap": why traditional ASR fails in ICPS

While the paradigms discussed above have shown promise in isolated software environments, applying them to Distributed ICPS reveals a fundamental "Distributed Gap." In a monolithic application, a bug fix is a local code transformation $P \rightarrow P'$. However, in a distributed ICPS, a repair action is a complex orchestration involving time, state, and topology. The challenges are no longer just about generating correct syntax, but about preserving system-wide invariants during and after the repair process.

We categorize these unique challenges into three critical dimensions, differentiating them from standard software repair tasks:

- **Cross-node state consistency repair:** In distributed ICPS, an SFB often leaves the system in a "corrupted state" distributed across multiple nodes. Merely patching the firmware code on the faulty node is insufficient if the distributed state remains inconsistent (The State Residual Problem). Furthermore, a repair strategy must ensure Global Repair Atomicity to prevent "Split-Brain" scenarios where repairing one node while others process data breaks system invariants.
- **Protocol semantic repair:** SFBs often manifest as violations of protocol semantics (e.g., QoS mismatches) rather than internal errors. Distributed-Native bugs frequently arise from implicit assumption mismatches between heterogeneous vendors. In cases where source code is immutable, repair implies synthesizing an external Protocol Adapter to intercept and mutate packets, a capability missing in current source-code focused ASR paradigms.
- **Patch deployment under timing constraints:** ICPS requires high availability, making the "Stop-Patch-Restart" model unacceptable. Repairs must perform Hot-Patching within the idle time of a single control cycle to avoid "Control Jitter." Additionally, identifying a "Quiescent State" to safely apply semantic repairs without disrupting the continuous manufacturing process is a complex control-theoretic problem.

■ 6.2.7 The common ailment: paradigm failure of APR in distributed ICPS

Having systematically analyzed the mainly strategies of ASR, their common and fundamental obstacles in addressing SFB issues within distributed ICPS become unequivocally clear.

- **Absence of a reliable oracle.** All the ASR methods, implicitly or explicitly, rely on an "oracle"–typically a test suite–to validate the correctness of a patch. For a distributed, emergent defect triggered by a confluence of timing, state, and network interactions, crafting a stable, reproducible, unit-level test case is practically impossible. The absence of this oracle renders any APR paradigm that depends on a "test-and-validate" loop fundamentally inapplicable.
- **Mismatch in repair scope.** The unit of operation for all existing APR tools is the code text of a single program. The "lesion" of a distributed SFB, however, often lies not within the internal logic of any single component, but in the interaction protocols or timing relationships between components. Applying a code-level patch to one component may not only fail to solve the system-level issue but could also violate a system-wide behavioral invariant, triggering a more catastrophic cascading failure elsewhere. The unit of repair does not align with the unit of failure.
- **Semantic disconnect in ICPS contexts.** The semantics

learned by ASR models are those of general-purpose programming languages. They do not comprehend the semantics of industrial protocols (e.g., the state machine of Modbus), the constraints of a Real-Time Operating System (RTOS), or, most importantly, the physical processes they control. A "fix" that appears correct at the code level–such as accelerating a response to avoid a timeout–could, in the physical world, translate to a valve closing too rapidly, inducing a "water hammer effect" and causing permanent physical damage. This ignorance of physical safety semantics represents an insurmountable gap for existing APR paradigms.

In summary, the current automatic repair paradigms, due to their reliance on a test oracle, their mismatched scope of repair, and their semantic ignorance of cyber-physical systems, are ineffective for resolving SFB issues in distributed ICPS. This lays the foundational necessity and novelty for the system-behavior-oriented analysis and specification methodology proposed in the remainder of this paper.

7 Future works and directions

Our paradigm-level analysis in Section 6 systematically demonstrated why mainstream detection and repair methods encounter a general "paradigm failure" when confronted with the complexities of distributed ICPS. This finding leads us directly to our final research question (RQ4): *What are the key challenges and future research directions for enhancing SFB security in distributed ICPSs*? To answer this, this section first synthesizes the preceding analysis to crystallize three fundamental challenges: the Representation Gap, the Black-Box Dilemma, and the Oracle Problem. Then, we propose a comprehensive, long-term vision for a new, system-behavior-oriented security framework designed specifically to overcome these challenges, providing a blueprint for the next generation of security assurance systems.

Through a paradigm-level analysis of mainstream security solutions in section 6, we systematically demonstrated why the mainstream detection and repair methods proven effective in traditional, single device systems encounter a general "paradigm failure" when confronted with the complexities of distributed ICPS. Our analysis revealed three fundamental challenges rooted in the nature of distributed systems:

- Representation gap: Existing models are confined to analyzing the internal logic of a single program, failing to effectively model the system-level interactive behaviors that span across components and are asynchronous and dependent on network timing.
- Black-box dilemma: A significant number of critical firmware in distributed ICPS exist as binary "black boxes". This lack of source code makes deep and precise semantic analysis nearly impossible, posing the primary obstacle to achieving system-level analysis.
- Oracle problem: For SFBs that emerge from multi-node interactions, creating a test case (a "dynamic oracle") that can reliably reproduce the defect is practically impossible. This renders all automatic repair methods that rely on a "test-and-validate" loop fundamentally ineffective.

To address the limitations identified in current research, specifically the Black-Box Dilemma, the Representation Gap, and the Oracle Problem, we propose a strategic and phased roadmap for the evolution of resilient distributed ICPS. Instead of seeking a single silver-bullet solution, we break down the vision into short-term (1−2 years), medium-term (3−5 years), and long-term (5−10 years) goals. This roadmap guides the transition from component-level analysis to autonomous system-level resilience.

7.1 Phase I (1–2 years): component transparency & semantic lifting

The immediate goal for the next 1−2 years is to overcome the Black-Box Dilemma by establishing a reliable pipeline for semantic recovery. To achieve a system-wide behavioral analysis of a distributed system, the first priority is to solve the "black-box dilemma" we defined in the introduction. In distributed ICPS, a common and serious challenge is the diversity and lack of transparency in their device firmware. Many key functions are handled by legacy systems, third-party proprietary libraries, or embedded device firmware, which are usually available only as binary files. The source code for these components is often unavailable due to business secrecy, outdated technology, or intellectual property restrictions. This lack of transparency at the code level makes it hard to use traditional static analysis for security, leaving hidden flaws (SFBs) in these binaries as a major blind spot for system safety and reliability.

To tackle this problem, we propose a three-stage automated analysis pipeline centered around a Large Language Model (LLM), aiming to recover the core program logic from binary files and transform the hidden information in these "black-box" binaries into a standardized input for deep analysis. The implementation path is detailed as follows:

Specifically, our strategy is structured as a three-stage automated analysis pipeline, with the implementation path detailed as follows:

- *Stage 1*: *Standardized decompilation*: As the starting point of the analysis, this stage employs mature industry-standard reverse engineering frameworks (e.g., NSA's Ghidra or Hex-Rays' IDA Pro) to decompile the target binary file. This process converts the machine code instruction set into a human-readable assembly language representation. However, while assembly code is precise, its granularity is too fine and lacks the abstraction of high-level program structures. Directly analyzing complex interaction logic and subtle algorithmic defects using assembly code is not only inefficient but also highly error-prone. Therefore, the output of this stage primarily serves as reliable input for the semantic enhancement in the next stage.
- *Stage 2*: *LLM-based semantic lifting of program logic*: This stage represents the core innovation of our methodology. We leverage a Large Language Model (LLM) pre-trained on vast

multilingual code repositories and specifically fine-tuned with instructions for the task of "assembly-to-high-level-language" conversion. The model takes the assembly code generated in the first stage as input and performs a "semantic lifting" task. The goal is not to achieve 100% source code recovery but to faithfully reconstruct the intrinsic logic of the program. Specifically, the LLM is trained to recognize low-level instruction patterns and map them to high-level language constructs. For example, it identifies instruction sequences related to stack pointer operations to reconstruct function calls and local variables; analyzes jump instructions to rebuild if-else, switch-case, and loop structures; and infers basic data structures based on memory access patterns. The final output of this stage is pseudo-source code in a high-level language format (e.g., pseudo-C/C++), which preserves the original program's key control flow graph (CFG), data dependencies, and core algorithms.

- *Stage 3*: *Learning-based semantic enrichment for SFB representation and detection*: Building on the advantages of semantic learning-based detection for identifying SFBs in firmware, as discussed earlier, our strategy continues to leverage the deep learning paradigm. The core of this approach lies in integrating semantic understanding with structural analysis of the code. Specifically, we parse the pseudo-source code into graph representations (e.g., Control Flow Graph (CFG), Data Flow Graph (DFG), Local Semantic Flow Graph (LSFG), or Code Property Graph (CPG)). A pre-trained code language model (e.g., CodeBERT) is then used to generate context-rich semantic embeddings for each node in the graph (representing variables, function calls, etc.), enabling the model to deeply understand "what the code fragment represents". These semantic embeddings serve as node features and are fed into a GNN. The GNN learns the topological structure of the graph—i.e., the control and data flows of the code—to capture "how different semantic fragments interact". Through this approach, the model can automatically learn and identify code patterns highly correlated with SFBs, defined by specific semantics and complex structures, achieving more accurate and generalizable automated detection compared to traditional methods.

In summary, the ultimate goal of this three-stage pipeline is not merely to find SFBs within a single binary file, but to produce a high-quality pseudo-code and its graph representation (e.g., a CPG) that preserves key semantics. This output will serve as the cornerstone for our next phase of building a "system-wide behavioral analysis framework", thereby paving the way to truly address the "representation gap" challenge at the distributed system level. A critical milestone in this phase will be mitigating the hallucination of LLMs to ensure that the recovered semantics faithfully reflect the original firmware logic.

7.2 Phase II (3–5 years): systemic modeling via SCPG

In the medium term (3−5 years), the focus shifts from individual nodes to the connections between them, aiming to bridge the Representation Gap. The process mentioned above can essentially be regarded as a form of semantic information enhancement for the code within the device firmware, thereby providing the capability to analyze the internal logic of individual "black-box" components. However, this strategy only addresses the challenge of detecting SFBs within distributed ICPS device firmware. The real challenge lies in understanding how these independent components, each with its own complex operational mechanisms, interact with one another through asynchronous messages, shared states, and intricate network protocols to form a whole whose behavior is difficult to predict. The fundamental limitation of existing analysis paradigms is that their "field of view" is confined within a single component, preventing them from observing this critically important "global interaction landscape". Therefore, to directly confront the representation gap challenge, we must conceive of a new framework capable of uniformly representing and analyzing the entire distributed system.

At the core of our vision is a new representation that we call the system-wide code property graph (SCPG). An SCPG is not a CPG of a single program, but rather a heterogeneous huge graph formed by "stitching together" the CPGs of all components in the system, whether derived from original source code or from pseudo-code recovered from binaries using the method in Section VII.A. The key to this "stitching" is the introduction of "interaction edges" to explicitly model inter-component communication. For instance, a node representing a call to *mqtt_publish*("*topic_A*",...) in one component can be connected via a "publish-subscribe" interaction edge to the entry node of the *on_message*("*topic_A*",...) callback handler in another component. Similarly, an RPC client call can be directly linked to the corresponding server-side implementation. In this way, the SCPG, for the first time at the model level, unifies the intra-program logic flow (control and data flow) with the inter-program interaction flow (network communication), providing a structural foundation for end-to-end system behavior tracing and analysis.

With the SCPG data structure, we can then perform comprehensive semantic information analysis upon it. We propose a hybrid analysis model:

- A GNN is continually responsible for learning the vast and complex topological structure of the SCPG. The ability of GNNs to capture long-range dependencies within a graph makes them ideal for modeling complex causal chains that span multiple physical devices and network layers, such as the entire closed-loop control flow from a sensor's data generation, through an edge gateway's processing, to a cloud AI's decision, and finally back to a physical actuator.
- A LLM provides deep semantic information understanding for each node (e.g., code blocks, API calls) within the SCPG. It should be noted that in this context, a node represents a unit derived from the entire code of a device's firmware, which is different with the graph representation mentioned above. The LLM, with its powerful pre-trained knowledge, answers the question, "What is the intent of this piece of code?"

When the GNN's ability to learn "how behavior occurs (structure)" is combined with the LLM's ability to understand "the meaning of each behavioral step (semantic information)", it is anticipated that a model designed under this strategy will be able to comprehend complex, high-level system events such as "component A sent an emergency shutdown signal to Component B". Instead of being trained on labeled "buggy" examples, this hybrid model is trained on a vast dataset of SCPGs representing normal, healthy system operations. Its primary task is to learn a deep, probabilistic model of this "normal behavior space".

During analysis, when a new system's behavior is presented as an SCPG, the model calculates the likelihood of this behavior occurring based on its learned knowledge. We can then derive a quantitative "Anomaly Score" from this likelihood. A behavior that is common and expected within the normal operational data will receive a low score, while a rare, unexpected, or logically inconsistent interaction, which is the hallmark of a system-level SFB, will be assigned a high Anomaly Score due to its low probability.

Ultimately, this framework shifts the focus of SFB detection. The objective is no longer to find specific, known vulnerable code, but to identify and quantify system-level behavioral deviations. A high Anomaly Score not only flags a problem but also provides a metric for its severity. Furthermore, by analyzing which parts of the SCPG contribute most to the high score, the model can automatically pinpoint the "anomalous subgraph". This subgraph, along with its score, provides the precise context needed for the automated repair process discussed next. The deep challenge here lies in scalability: developing graph learning algorithms capable of handling dynamic topologies in large-scale industrial networks without compromising real-time performance.

7.3 Phase III (5–10 years): autonomous resilience & trustworthy evolution

Looking towards the long term (5−10 years), the vision is to solve the Oracle Problem and achieve a self-healing ICPS that requires minimal human intervention. The detection strategy proposed herein offers a novel perspective on SFB detection. However, a complete solution must not only identify problems but also guide their remediation. Therefore, our long-term vision is to construct a framework capable of leveraging this diagnostic information to proactively guide the bug-fixing process. The core lies in a fundamental paradigm shift: transforming the "anomaly score" produced from a static diagnostic metric into a dynamic, optimizable objective for repair. We term this the guiding principle of anomaly reduction.

Under this principle, the success of a repair is no longer contingent on the subjective assessment of human experts or predefined rule sets. Instead, it is defined by a clear, quantifiable metric: an effective repair must demonstrably and significantly lower the system's Anomaly Score. In essence, our goal is to steer the system's behavioral state, as represented by its SCPG, from a high-score anomaly region back into the low-score space of normal operations through precise modifications. This principle provides the most fundamental direction and measure of success for the automated repair process that follows.

With this guiding principle established, we next envision a powerful core engine capable of achieving this objective. We propose leveraging a generative model, specifically a foundational model trained on code and system interaction logic, to serve as the generator for automated repair solutions.

The workflow for this repair engine is as follows: first, it takes the "anomalous subgraph" identified in Section B as a precise, contextual input, allowing the model to focus on the root of the problem. Then, guided by the core directive to "minimize the anomaly score", the generative model does not engage in blind trial-and-error but instead purposefully explores modifications that can return the system's behavior to a state of normalcy. It may generate multiple candidate "patches," which could be subtle adjustments to component code, configuration files, or their interaction protocols.

Crucially, every candidate patch generated by the model enters a virtual, closed-loop evaluation process. We would virtually apply the patch to the original SCPG, creating a new, hypothetical SCPG post-repair. Subsequently, this new SCPG is fed back into the anomaly detection model from Section B to calculate a predicted new anomaly score. This process transforms the abstract task of "repair" into a concrete, quantifiable optimization problem: finding the modification that maximizes the reduction in the anomaly score. Research must evolve from simple patching to "trustworthy autonomy", where AI-generated repairs are formally verified before deployment to guarantee safety in critical physical processes.

8 Conclusion

In this article, we examine the critical role of ICPS in modern industry, focusing on the increasing importance of cloud and fog computing in distributed ICPS architectures. We highlighted the growing security concerns related to SFBs in devices within these distributed DFC-ICPS environments, emphasizing how these bugs can be exploited to compromise operational safety and data security. By investigating the root causes of SFBs in DFC-ICPS and discussing the resultant security implications, we aim to raise awareness of these SFBs. Furthermore, the taxonomy and cascading failure model proposed in this study offer practical utility beyond theoretical analysis; the proposed analytical framework can serve as a reference for industrial firmware security inspection and the development of more resilient distributed protocols. Finally, we proposed potential solutions and future research directions to strengthen the security posture of DFC-ICPS and mitigate the risks posed by SFBs, paving the way for more secure and reliable industrial control systems.

Acknowledgments

This work was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project CityU 11217124, the National Natural Science Foundation of China (Grant No. 62472218), and the Peng Cheng Laboratory Project (PCL2025A13).

Competing interests

The authors declare that they have no competing interests or financial conflicts to disclose.

Open Access

This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made.

The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

To view a copy of this licence, visit website at creativecommons.org/licenses/by/4.0/.

References

[1] Zhang K, Shi Y, Karnouskos S, Sauter T, Fang H, Colombo A W. Advancements in industrial cyber-physical systems: an overview and perspectives. IEEE Transactions on Industrial Informatics, 2023, 19(1): 716−729

[2] Kayan H, Nunes M, Rana O, Burnap P, Perera C. Cybersecurity of industrial cyber-physical systems: a review. ACM Computing Surveys (CSUR), 2022, 54(11s): 229

[3] Qian L, Luo Z, Du Y, Guo L. Cloud computing: an overview. In: Proceedings of the 1st International Conference on Cloud Computing. 2009, 626−631

[4] Yi S, Li C, Li Q. A survey of fog computing: concepts, applications and issues. In: Proceedings of 2015 Workshop on Mobile Big Data. 2015, 37−42

[5] Makhshari A, Mesbah A. IoT bugs and development challenges. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE). 2021, 460−472

[6] Farhadi M, Lanet J L, Pierre G, Miorandi D. A systematic approach toward security in Fog computing: assets, vulnerabilities, possible countermeasures. Software: Practice and Experience, 2020, 50(6): 973−997

[7] Zhou W, Cao C, Huo D, Cheng K, Zhang L, Guan L, Liu T, Jia Y, Zheng Y, Zhang Y, Sun L, Wang Y, Liu P. Reviewing IoT security via logic bugs in IoT platforms and systems. IEEE Internet of Things Journal, 2021, 8(14): 11621−11639

[8] Kuhrmann M, Fernández D M, Daneva M. On the pragmatic design of literature studies in software engineering: an experience-based guideline[J]. Empirical software engineering, 2017, 22(6): 2852−2891

[9] Petersen K, Feldt R, Mujtaba S, Mattsson M. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE). 2008, 68−77

[10] Lee J, Bagheri B, Kao H A. A cyber-physical systems architecture for industry 4. 0-based manufacturing systems. Manufacturing Letters, 2015, 3: 18−23

[11] Monostori L, Kádár B, Bauernhansl T, Kondoh S, Kumara S, Reinhart G, Sauer O, Schuh G, Sihn W, Ueda K. Cyber-physical systems in manufacturing. CIRP Annals, 2016, 65(2): 621−641

[12] Satchidanandan B, Kumar P R. Dynamic watermarking: active defense of networked cyber–physical systems. Proceedings of the IEEE, 2017, 105(2): 219−240

[13] Ding D, Han Q L, Xiang Y, Ge X, Zhang X M. A survey on security control and attack detection for industrial cyber-physical systems. Neurocomputing, 2018, 275: 1674−1683

[14] Leitão P, Colombo A W, Karnouskos S. Industrial automation based on cyber-physical systems technologies: prototype implementations and challenges. Computers in Industry, 2016, 81: 11−25

[15] Zampetti F, Kapur R, Di Penta M, Panichella S. An empirical characterization of software bugs in open-source cyber–physical systems. Journal of Systems and Software, 2022, 192: 111425

[16] Patra J, Pradel M. Semantic bug seeding: a learning-based approach for creating realistic bugs. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021, 906−918

[17] Khatiwada S, Tushev M, Mahmoud A. Just enough semantics: an information theoretic approach for IR-based software bug localization. Information and Software Technology, 2018, 93: 45−57

[18] Chen X, Zhu H, Cao J, Wen M, Cheung S C. SemBIC: semantic-aware identification of bug-inducing commits. Proceedings of the ACM on Software Engineering, 2025, 2(FSE): 1363−1385

[19] Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C. Bug characteristics in open source software. Empirical Software Engineering, 2014, 19(6): 1665−1705

[20] Catolino G, Palomba F, Zaidman A, Ferrucci F. Not all bugs are the same: understanding, characterizing, and classifying bug types. Journal of Systems and Software, 2019, 152: 165−181

[21] Grätz L, Hahnle R, Bubel R. Finding semantic bugs fast. In: Proceedings of the 25th International Conference on Fundamental Approaches to Software Engineering. 2022, 145−154

[22] Lion B, Arbab F, Talcott C. A formal framework for distributed cyber-physical systems. Journal of Logical and Algebraic Methods in Programming, 2022, 128: 100795

[23] Cheng L, Liu F, Yao D. Enterprise data breach: causes, challenges, prevention, and future directions. WIREs Data Mining and Knowledge Discovery, 2017, 7(5): e1211

[24] Liu K, Koyuncu A, Kim D, Bissyandè T F. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019, 1−12

[25] Kim M, Stehr M O, Talcott C. A distributed logic for networked cyber-physical systems. Science of Computer Programming, 2013, 78(12): 2453−2467

[26] Rotem-Gal-Oz A. Microservices granularity tradeoffs. See rgoarchitects.com/Files/fallacies website, 2025

[27] Wu H, Pan J, Huang P. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In: Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 2024, 70

[28] Fang C, Liang J, Wu Z, Fu J, Jia Z, Huang C, Jiang Y, Li S. DepState: detecting synchronization failure bugs in distributed database management systems. Proceedings of the ACM on Software Engineering,

2025, 2(ISSTA): ISSTA088

[29] Zou Y H, Bai J J, Zhou J, Tan J, Qin C, Hu S M. TCP-Fuzz: detecting memory and semantic bugs in TCP stacks with fuzzing. In: Proceedings of 2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021, 489−502

[30] Nilizadeh A, Leavens G T, Păsăreanu C S, Noller Y. JMLKelinci+: detecting semantic bugs and covering branches with valid inputs using coverage-guided fuzzing and runtime assertion checking. Formal Aspects of Computing, 2024, 36(1): 2

[31] Soldani J, Brogi A. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: a survey. ACM Computing Surveys (CSUR), 2023, 55(3): 59

[32] Sasi T, Lashkari A H, Lu R, Xiong P, Iqbal S. A comprehensive survey on IoT attacks: taxonomy, detection mechanisms and challenges. Journal of Information and Intelligence, 2024, 2(6): 455−513

[33] Guo F, Yu F R, Zhang H, Li X, Ji H, Leung V C M. Enabling massive IoT toward 6G: a comprehensive survey. IEEE Internet of Things Journal, 2021, 8(15): 11891−11915

[34] Huang J, Liu F, Zhang J. Multi-dimensional QoS evaluation and optimization of mobile edge computing for IoT: a survey. Chinese Journal of Electronics, 2024, 33(4): 859−874

[35] Lou C, Huang P, Smith S. Understanding, detecting and localizing partial failures in large system software. In: Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). 2020, 559−574

[36] Samonas S, Coss D. The CIA strikes back: redefining confidentiality, integrity, and availability in security. Journal of Information System Security, 2014, 10(3): 21−45

[37] Chandramohan M, Xue Y, Xu Z, Liu Y, Cho C Y, Tan H B K. BinGo: cross-architecture cross-OS binary search. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, 678−689

[38] Hu Y, Zhang Y, Li J, Gu D. Binary code clone detection across architectures and compiling configurations. In: Proceedings of the 25th IEEE/ACM International Conference on Program Comprehension (ICPC). 2017, 88−98

[39] Bilge L, Dumitraş T. Before we knew it: an empirical study of zero-day attacks in the real world. In: Proceedings of 2012 ACM Conference on Computer and Communications Security. 2012, 833−844

[40] Takanen A, Demott J, Miller C, et al. Fuzzing for Software Security Testing and Quality Assurance. Boston: Artech House, 2018

[41] Chesser M, Nepal S, Ranasinghe D C. MultiFuzz: a multi-stream fuzzer for testing monolithic firmware. In: Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24). 2024, 5359−5376

[42] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: a fast address sanity checker. In: Proceedings of 2012 USENIX Conference on Annual Technical Conference. 2012, 28

[43] Shiri Harzevili N, Boaye Belle A, Wang J, Wang S, Jiang Z, Nagappan N. A systematic literature review on automated software vulnerability detection using machine learning. ACM Computing Surveys, 2025, 57(3): 55

[44] Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H. Scalable graph-based bug search for firmware images. In: Proceedings of 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016, 480−491

[45] Gao D, Reiter M K, Song D. BinHunt: automatically finding semantic differences in binary programs. In: Proceedings of the 10th International Conference on Information and Communications Security. 2008, 238−255

[46] Ming J, Pan M, Gao D. iBinHunt: binary hunting with inter-procedural control flow. In: Proceedings of the 15th International Conference on Information Security and Cryptology. 2012, 92−109

[47] Xu X, Liu C, Feng Q, Yin H, Song L, Song D. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017, 363−376

[48] Ribeiro L F R, Saverese P H P, Figueiredo D R. *struc2vec*. Learning node representations from structural identity. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2017, 385−394

[49] Zhou Y, Liu S, Siow J, Du X, Liu Y. *Devign*: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. 2019, 915

[50] Gao J, Yang X, Fu Y, Jiang Y, Sun J. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018, 896−899

[51] Gao J, Jiang Y, Liu Z, Yang X, Wang C, Jiao X, Yang Z, Sun J. Semantic learning and emulation based cross-platform binary vulnerability seeker. IEEE Transactions on Software Engineering, 2021, 47(11): 2575−2589

[52] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA). 2018, 757−762

[53] Liu J, Zeng J, Wang X, Liang Z. Learning graph-based code representations for source-level functional similarity detection. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 345−357

[54] Liu B, Huo W, Zhang C, Li W, Li F, Piao A, Zou W. αDiff: cross-version binary code similarity detection with DNN. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE). 2018, 667−678

[55] Wu Y, Zou D, Dou S, Yang W, Xu D, Jin H. VulCNN: an image-inspired scalable vulnerability detection system. In: Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE). 2022, 2365−2376

[56] Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y. VulDeePecker: a deep learning-based system for vulnerability detection. In: Proceedings of the 25th Network and Distributed System Security Symposium. 2018

[57] Schuster M, Paliwal K K. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 1997, 45(11): 2673−2681

[58] Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE

Transactions on Dependable and Secure Computing, 2022, 19(4): 2244−2258

[59] Yu Z, Cao R, Tang Q, Nie S, Huang J, Wu S. Order matters: semantic-aware neural networks for binary code similarity detection. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence. 2020, 1145−1152

[60] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016, 770−778

[61] Hoang T D, Dam H K, Kamei Y, Lo D, Ubayashi N. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). 2019, 34−45

[62] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017, 6000−6010

[63] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of the Association for Computational Linguistics: EMNLP 2020. 2020, 1536−1547

[64] Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng S K, Clement C B, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M. GraphCodeBERT: pre-training code representations with data flow. In: Proceedings of the 9th International Conference on Learning Representations. 2021

[65] Guo D, Lu S, Duan N, Wang Y, Zhou M, Yin J. UniXcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2022, 7212−7225

[66] Du X, Wen M, Zhu J, Xie Z, Ji B, Liu H, Shi X, Jin H. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In: Proceedings of the Association for Computational Linguistics: ACL 2024. 2024, 10507−10521

[67] Liu H, Liu Y, Jiang Y, Zhang Z, Chen L, Hancke G P. Cross-device firmware bug detection in deployed IoT systems based on twisted-CodeBERT-GCN semantic learning. IEEE Transactions on Consumer Electronics, 2026, 72(1): 619−637

[68] Lekssays A, Mouhcine H, Tran K, Yu T, Khalil I. LLMxCPG: context-aware vulnerability detection through code property graph-guided large language models. In: Proceedings of the 34th USENIX Security Symposium (USENIX Security 25). 2025, 26

[69] Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. In: Proceedings of 2009 31st International Conference on Software Engineering. 2009, 364−374

[70] Qi Y, Mao X, Lei Y, Dai Z, Wang C. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 254−265

[71] Tan S H, Yoshida H, Prasad M R, Roychoudhury A. Anti-patterns in search-based program repair. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, 727−738

[72] Nguyen H D T, Qi D, Roychoudhury A, Chandra S. SemFix: program repair via semantic analysis. In: Proceedings of the 35th International Conference on Software Engineering (ICSE). 2013, 772−781

[73] Mechtaev S, Yi J, Roychoudhury A. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 691−701

[74] Xuan J, Martinez M, Demarco F, Clément M, Marcote S L, Durieux T, Berre D L, Monperrus M. Nopol: automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering, 2017, 43(1): 34−55

[75] Zhang J, Cambronero J P, Gulwani S, Le V, Piskac R, Soares G, Verbruggen G. PyDex: repairing bugs in introductory python assignments using LLMs. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA1): 1100−1124

[76] Zhou X, Xu B, Kim K, Han D, Nguyen H H, Le-Cong T, He J, Le B, Lo D. Leveraging large language model for automatic patch correctness assessment. IEEE Transactions on Software Engineering, 2024, 50(11): 2865−2883

[77] Gupta R, Pal S, Kanade A, Shevade S. DeepFix: fixing common c language errors by deep learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence. 2017, 1345−1351

[78] Lutellier T, Pham HV, Pang L, Li Y, Wei M, Tan L. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020, 101−114

[79] Chen Z, Kommrusch S, Tufano M, Pouchet L N, Poshyvanyk D, Monperrus M. SequenceR: sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 2021, 47(9): 1943−1959

[80] Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto H P, et al. Evaluating large language models trained on code. 2021, arXiv preprint arXiv: 2107.03374

[81] OpenAI. GPT-4 technical report. 2023, arXiv preprint arXiv: 2303.08774

[82] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: end-to-end program repair with LLMs. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023, 1646−1656

[83] Silva A, Fang S, Monperrus M. RepairLLaMA: efficient representations and fine-tuned adapters for program repair. IEEE Transactions on Software Engineering, 2025, 51(8): 2366−2380

[84] Monperrus M. Automatic software repair: a bibliography. ACM Computing Surveys (CSUR), 2019, 51(1): 17

[85] Hayden C M, Smith E K, Denchev M, Hicks M, Foster J S. Kitsune: efficient, general-purpose dynamic software updating for C. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. 2012, 249−264

[86] Hicks M, Moore J T, Nettles S. Dynamic software updating. ACM SIGPLAN Notices, 2001, 36(5): 13−23

[87] Noura M, Atiquzzaman M, Gaedke M. Interoperability in internet of things: taxonomies and open challenges. Mobile Networks and Applications, 2019, 24(3): 796−809

[88] Seol Y, Hyeon D, Min J, Kim M, Paek J. Timely survey of time-sensitive networking: past and future directions. IEEE Access, 2021, 9:

142506−142527

[89] Zhan X, Wu W, Shen L, Liao W, Zhao Z, Xia J. Industrial internet of things and unsupervised deep learning enabled real-time occupational safety monitoring in cold storage warehouse. Safety Science, 2022, 152: 105766



Hao LIU is currently pursuing the PhD degree with the School of Cyber Science and Engineering, Southeast University, China and Department of Computer Science, City University of Hong Kong, China. His research interests include IoT firmware bug detection and automatic repair. He was also a research assistant in Zhejiang Lab, China, from 2023 to 2024.



Yucheng LIU received the BEng degree in electronic and information engineering from Dalian University, China in 2016, and the master's degree in electronic and information engineering in 2017 from the City University of Hong Kong (CityUHK), China. Then, he finished his PhD study in Department of Electrical Engineering of CityUHK (2023). After that, he worked as postdoc in Department of Computer Science of CityUHK (2023−2024). He is now the Assistant Research Fellow in Pengcheng Lab, China. His current research interests include embedded system, industrial IoT, cybersecurity, and standard.



Bruno SILVA received the BEng, MEng, and PhD degrees in computer engineering from the University of Pretoria, South Africa in 2011, 2015, and 2021, respectively. From September 2013 to February 2014, he was a visiting researcher with the ABB Corporate Research Centre, Sweden. From November 2015 to November 2018, he was a junior research officer with the University of Pretoria. He is currently a research fellow with the City University of Hong Kong, China. His research interests include indoor localization and mobile computing.



Raphael E. NKROW received BSc degree in computer science from the Kwame Nkrumah University of Science and Technology (KNUST), Ghana in 2014, and MEng degree in information and communication engineering from the University of Electronic Science and Technology of China (UESTC), China in 2019. He completed his PhD in computer science at the City University of Hong Kong, China in 2025, where he is currently a postdoctoral fellow. His research interests include indoor localization, machine learning, transfer adaptation learning, UWB distance-bounding protocols, and the security of wireless ranging systems.



Dutliff BOSHOFF received BSc degree in electrical engineering from City University of Hong Kong (CityUHK) , China in 2021 and PhD degree in computer science from CityUHK in 2025. He is currently a postdoctoral fellow in the Department of Data Science and serves as Tech. lead at Streli Technology Limited, a university spin-off. His research interests include UWB physical-layer key sharing, UWB ranging and distance-bounding protocols, and the application of LLM-based agents for industrial process optimization.



Lu ZHOU is a professor of School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China and is a doctoral supervisor, a national-level young talent, and the director of the Isec Laboratory, a key laboratory of the Ministry of Industry and Information Technology. Her main research interests include artificial intelligence security, blockchain applications, and cryptographic engineering. She has published over 80 papers in internationally significant security and cryptography academic journals and conferences. Her research is supported by over 10 national/provincial/ministerial-level projects. She has published one textbook named "Cryptography Engineering", and participated in the development of the group standard for "Blockchain Commodity Warehouse Receipt Data Specification". She serves as a young editorial board member for multiple domestic and international security-related journals, including Q1 SCI journal BDMA, Journal of Cyberspace Security Science, and Journal of Network and Information Security.



Liquan CHEN received the PhD degree from Southeast University, China in 2005. He was a visiting scholar at the National University of Singapore, Singapore from 2011 to 2012. He is currently a professor with the School of Cyber Science and Engineering, Southeast University, China. His research interests include information security, cryptography, and network security.



Gerhard P. HANCKE is a professor in the Department of Computer Science at City University of Hong Kong, China. He received the BEng and MEng degrees in computer engineering from the University of Pretoria, South Africa in 2002 and 2003, and the PhD degree in computer science from the University of Cambridge, UK in 2009. Previously he worked as a researcher with the Smart Card and IoT Security Centre and as teaching fellow with the Department of Information Security, both at Royal Holloway, University of London, UK. His research interests in security, reliable communication, and distributed sensing for the Industrial Internet-of-Things.