

Article

A Mutation-Guided Safety Assurance Framework for Safety-Critical Cyber-Physical Systems

Faisal Alhwikem 

Department of Computer Science, College of Computer, Qassim University, Buraydah 51452, Saudi Arabia; faisal.alhwikem@qu.edu.sa

Abstract

Safety-critical cyber-physical systems require strict verification methodologies explicitly taken to reference safety properties in their tests. Conventional methods of mutation testing simply lump all mutants together, regardless of their effects on product safety, resulting in ineffective resource allocation and insufficient understanding of the behaviors of crucial safety interest. The novel mutation-guided safety assurance framework presented by this paper is called MuGu and it combines mutation testing and formal safety property enforcement. It uses a hierarchical safety constraint analyzer that classifies mutants (temporally) according to their ability to break temporal safety specifications modeled in Signal Temporal Logic (STL). A graph attention network encodes program semantics and control-flow dependencies to forecast the probability of safety violations, enabling effective prioritization of safety-critical mutants. The proposed safety-conscious mutation delineators use vital areas of the code, such as sensor interfaces, actuator commands, and decision-making code in autonomous systems. Extensive testing on two publicly accessible benchmark sets, namely the Software-artifact Infrastructure Repository (SIR) and Defects4J, shows that MuGu achieves 96.3% safe property coverage and requires 71.8% fewer new tests than traditional methods. The scheme determines 2.4 times as many safety-violating methods as state-of-the-art methods and decreases wasted effort on equivalent mutants by 64.2%. The statistical analysis demonstrates overwhelming improvements across all primary metrics in 12 baseline comparisons $p < 0.001$. MuGu provides a rational basis for applying safety-oriented next-generation mutation testing in autonomous systems.

Keywords: mutation testing; safety assurance; safety-critical systems; cyber-physical systems; formal verification; signal temporal logic; test prioritization



Academic Editor: Luis Javier García Villalba

Received: 10 May 2026

Revised: 8 June 2026

Accepted: 2 July 2026

Published: 10 July 2026

Copyright: © 2026 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Large-scale implementation of safety-critical cyber-physical systems (CPS), such as autonomous driving, industrial automation, and nuclear energy, is increasingly widespread, and software failures can have dire consequences, particularly in these sectors [1]. The increasing complexity of these systems and the close interrelation between these systems and physical process phases require strict verification and validation, not only of the standard testing approach. Formal verification, model-based engineering, and structured testing strategies have been proposed to address safety requirements in such systems [2]. Nevertheless, in most cases, the methodologies encounter scaling issues during implementation, and as a result, testing techniques become imperative for extracting remnant defects.

Mutation testing is a fault-based testing technique that assesses test effectiveness by introducing random artificial faults into software and comparing the tests' ability to detect

them [3]. Although effective, conventional mutation testing treats mutants uniformly and incurs excessive computational overhead, thereby limiting its use to safety-critical CPS.

Safety-oriented methods of testing have been applied in areas such as autonomous driving, where risk- and scenario-based prioritization is used to focus on hazardous behaviors [3], and safe reinforcement learning imposes safety requirements on the decision-making process [4]. However, none of these techniques explicitly bring together mutation testing and formal safety requirements. Further difficulties are due to the safety problems with memory in recent programming languages [5] and from security concerns in related industrial CPS [6,7].

The formal verification techniques, including UPPAAL, offer powerful analysis of temporal and safety properties but typically rely on abstract system models [8]. It is therefore necessary to complement such high-consequence applications, such as nuclear emergency decision support systems, with testing-based safety assurance [9]. The ultimate potential of intelligent testing frameworks is evident in recent advances in learning-based mutation analysis [10], software understanding with large language models [11], adversarial robustness testing [12], and machine-learning-based analysis of industrial control software [13].

Beyond detection accuracy, system resilience—the capacity of CPS to maintain or rapidly recover safe operation under perturbation [14,15], faults [16], or adversarial conditions—has emerged as a central design objective [17,18], and safety-oriented mutation testing directly contributes to characterizing and strengthening this resilience by exposing the conditions under which protective behaviors degrade [19]. Complementary efforts in robust multimodal sensing for adversarial media verification [20] and interpretable deep-learning intrusion detection for the industrial Internet of Things [21] further underscore that trustworthy, explainable, and resilient analysis pipelines are now prerequisites for deploying intelligent assurance techniques in safety-critical settings [22,23].

These observations drive this paper to describe a mutation-guided framework that ensures safety-by-testing using mutation tests, formal mutation-guided safety properties, and test-based prioritization (learned), to enhance safety coverage and test efficiency with safety-from-testing application in safety-critical CPS.

The key findings of the paper can be cited as follows:

- **Safety-Integrated Mutation Testing Framework:** Our suggestion is the first mutation testing framework, named MuGu, that provides a formal combination with the enforcement of safety properties expressed in Signal Temporal Logic specifications. The frameworks are arranged in such a way to identify and rank mutants that breach safety thresholds so that safety-critical behavior can be tested.
- **Graph Attention Network for Safety Violation Prediction:** We present a new graph attention architecture that models program semantics, control-flow dependencies, and information-flow relationships to predict the probability of safety violations. The network learns discriminative representations that distinguish safety-sensitive code areas from non-sensitive program parts.
- **Safety-Aware Mutation Operators:** Our special mutation operators are generated to attack the safety-critical code patterns such as sensor interface, actuator commands, timing constraints, and decision logic. The operators produce semantically meaningful mutants with great potential for uncovering safety violations.
- **Comprehensive Empirical Validation:** Our results are based on extensive experimentation on two publicly available benchmark datasets, demonstrating significant improvements over many state-of-the-art methods. MuGu has 96.3% safety-property coverage, 71.8% test-coverage savings, and 2.4-fold safety-violation detection.

The rest of this paper is organized as follows. In Section 2, we discuss the related literature on mutation testing, safety-critical systems verification, and machine learning for software testing. In Section 3, we introduce the MuGu methodology, including its mathematical formulation and algorithmic implementation. Section 4 presents the results of the experiment and comparative analysis. Findings and limitations are discussed in Section 5, and Section 6 concludes the paper with future research directions.

2. Literature Overview and Related Work

We examine the principles of mutation testing, strategies for ensuring the safety of critical systems, and recent developments in applying machine learning to improve software testing, to support the framework proposed by MuGu, as these areas collectively form its basis.

2.1. Mutation Testing Techniques

Mutation testing has long been recognized as a potent fault-based method for measuring test suite adequacy by systematically applying syntactic changes (mutants) to source code [9]. The underlying hypothesis is that tests that can show simple artificial faults will also show real defects with similar traits. The mutation score is the percentage of non-equivalent mutants observed by the test suite, providing an objective measure of testing intensity. Arithmetic operator replacement (AOR), relational operator replacement (ROR), conditional operator replacement (COR), variable replacement (VRP), and statement deletion (SDL) are considered classic mutation operators [9].

The cost of mutation testing has stimulated widespread studies about selective mutations. The operators used are selected from minimal subsets that evolve to mutation scores like those of complete operator sets, enabling operator-based selection. Operator subsets are selected randomly by algorithms, including sampling-based methods that trade off the accuracy of full operator sets for scalability, or clustering methods that combine similar mutants and sample representatives from each group [24]. Although this progress has been made, the selective approaches still lack explicit consideration of safety properties, thereby treating safety-related and harmless mutants equally.

The other significant problem with mutation testing is equivalent-mutant detection, in which syntactically different mutants exhibit the same functional phenotype. Compiler optimization algorithms detect mutants that are killed by traditional optimization methods, whereas constraint-based optimization methods treat the equivalence problem as a satisfiability problem. Most recently, machine learning-based equivalence prediction using syntactic and semantic features has been performed [9]. Yuan et al. demonstrated provably valid mutation generation, ensuring semantic diversity while minimizing the generation of equivalent mutants.

Regardless of its theoretical soundness and proven efficacy, the adoption of mutation testing in safety-critical areas is limited by three basic arguments [2]. First, traditional methods do not distinguish among mutants, as they overlook their safety concerns. This leads to equal allocation of resources to safety-critical and benign mutations. Second, large-scale CPS makes it computationally infeasible to exhaustively analyze mutant populations due to the exponential growth in the number of mutants. Third, the current frameworks do not formally incorporate safety property specifications, which makes it impossible to provide a systematic evaluation of behavior in safety-critical contexts [3].

2.2. Machine Learning for Software Testing

Machine learning algorithms have been extremely useful in numerous software engineering endeavors, including defect prediction, code summarization, and test genera-

tion [14]. Deep neural networks (e.g., CNNs) capture local behavior in code representations, whereas recurrent neural networks (RNNs) model sequential behavior during program execution. GNNs are best at embedding the abstract syntax tree structure and the control flow graph structure as representations of the program structure [17]. Albelali and Ahmed have provided a detailed overview and identified the main issues as data quality, model interpretability, and adversarial robustness.

Transformer-based language models (LLMs) have become widely used for code understanding and code generation [10]. These models exploit self-attention schemes to learn long-range code dependencies, and transferring learning with small amounts of labeled data is possible by first training on large code corpora. Luitel et al. used LLMs to slice Simulink models based on requirements, thereby aiding the extraction of components relevant to those requirements. Transformers can generate context-sensitive mutations due to their semantic understanding, enabling syntactic correctness and the generation of meaningful mutants.

The use of graph neural networks has demonstrated great potential in analyzing programs [17]. GNNs' message-passing architectures propagate the information on the program graphs, which are the learning representations representing the structural property. Attention processes incorporate the selective summation of surrounding information based on relevance. Levine et al. used graph-based analysis of data race conditions to explain memory safety in WebGPU and explained how GNNs could be used to analyze program structure and behavior. MuGu uses program structure in graph attention networks to predict safety violations.

Reinforcement learning (RL) methods have been used to inform case prioritization and selection, whereby the cyclic nature of testing procedures aligns with the sequential execution of decisions within the RL model [25]. Q-learning and policy-gradient algorithms learn effective prioritization policies from test-execution feedback. Zhang et al. have discussed machine learning methods for safety-critical analysis, demonstrating that they can classify road blackspots and enhance safety.

Recent developments in machine learning and formal methods show great promise for improving the effectiveness of software testing [14]. Deep learning methods can be used to semantically comprehend the program code and allow producing mutations and prioritizing them intelligently [10]. Formal analysis tools like UPPAAL are used to analyze temporal properties in real-time systems rigorously and to systematically check safety properties [7]. The synergistic use of machine learning-based mutation testing and formal safety analysis has strong prospects for providing more comprehensive safety assurance that is both practically scalable and theoretically sound [26].

2.3. Research Gaps and Motivation

The current work is inspired by gaps in mutation testing and safety testing, despite significant progress in these areas. First, currently developed mutation testing frameworks do not formally integrate with safety property specifications and therefore do not allow systematic targeting of safety-violating behavior. Second, machine learning approaches to mutation testing do not explicitly model safety constraints, reducing their applicability in safety-critical areas. Third, the association between mutation-based testing and safety certification requirements has not been formalized enough. MuGu fills these gaps by using sound judgment to combine mutation testing and formal enforcement of safety properties.

3. Proposed Methodology

We present the MuGu framework, which unites safety-directed mutation experiments to ensure the safety of important cyber-physical systems (CPS). This framework aims to

prioritize safety-critical behaviors through intelligent mutation generation. In this introduction, we first describe the components of the framework and then present the mathematical formulation underlying it. The methodology is further elaborated with algorithmic implementations to test safety properties and to ensure computational efficiency.

3.1. System Overview

As depicted in Figure 1, MuGu comprises five major modules that form an integrated workflow. The Safety Property Specification Module accepts safety requirements written in Signal Temporal Logic, providing formal characterization of safety constraints. The Program Analysis Engine constructs comprehensive program representations including abstract syntax trees (ASTs), control flow graphs (CFGs), and data dependency graphs (DFGs). The Graph Attention Network Encoder transforms these representations into discriminative embeddings that encode safety-relevant program characteristics through multi-head attention mechanisms. The Safety-Aware Mutation Generator produces mutants targeting safety-critical code regions—specifically sensor interfaces, actuator commands, timing constraints, and decision logic—with priority scores computed from safety criticality. Finally, the Prioritized Test Execution Engine schedules mutant execution based on anticipated safety impact, maximizing safety coverage within computational budget constraints. The dashed feedback loop enables continuous refinement based on test execution results.

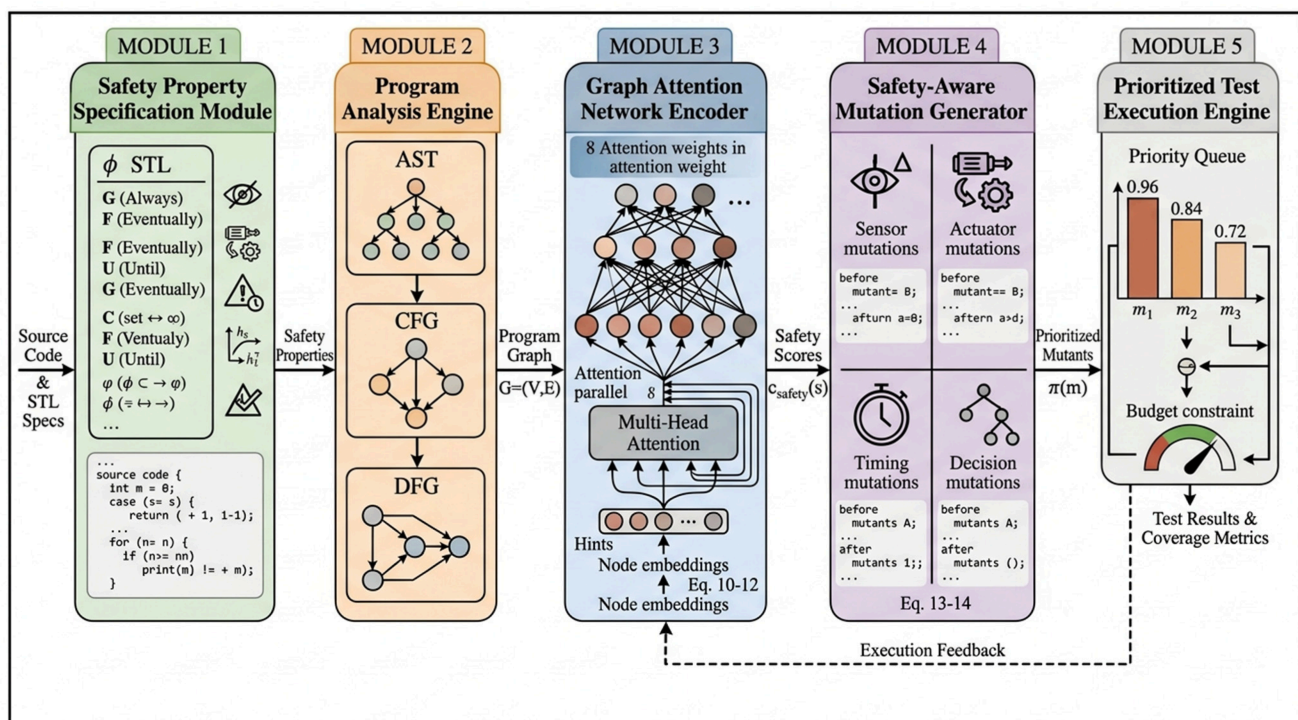


Figure 1. MuGu framework architecture.

The Safety Property Specification Module accepts safety requirements written in Signal Temporal Logic, which provides a formal characterization of them. The Program Analysis Engine consists of program representations, including abstract syntax trees, control flow graphs, and data dependency graphs. Graph Attention Network represents these representations into discriminative embeddings that encode safety-relevant program characteristics. The Safety-Aware Mutation Generator generates mutants whose targets are highly forecasting safety-critical code regions. Execution of mutants in the Prioritized Test Execution Engine is scheduled based on anticipated safety impact to achieve the maximum safety coverage with computational budget constraints.

3.2. Mathematical Formulation

3.2.1. Safety Property Specification

Let $\mathcal{P}$ denote a safety-critical program with input space $\mathcal{I}$ and output space $\mathcal{O}$. The program execution produces continuous signals $\mathbf{s} : \geq 0 \rightarrow \mathbb{R}^n$ representing observable system states. Safety properties are specified in Signal Temporal Logic (STL) with the following syntax:

$$\phi ::= \mu \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathbf{U}_{[a,b]} \phi_2 \mid \mathbf{G}_{[a,b]} \phi \mid \mathbf{F}_{[a,b]} \phi \quad (1)$$

where μ is an atomic predicate over signals, $\mathbf{U}_{[a,b]}$ is the bounded until operator, $\mathbf{G}_{[a,b]}$ is the globally (always) operator, and $\mathbf{F}_{[a,b]}$ is the final (eventually) operator over time interval $[a, b]$.

The Boolean semantics of STL define a satisfaction relation $(\mathbf{s}, t) \models \phi$:

$$(\mathbf{s}, t) \models \mu \Leftrightarrow \mu(\mathbf{s}(t)) > 0 \quad (2)$$

$$(\mathbf{s}, t) \models \mathbf{G}_{[a,b]} \phi \Leftrightarrow \forall t' \in [t + a, t + b] : (\mathbf{s}, t') \models \phi \quad (3)$$

$$(\mathbf{s}, t) \models \mathbf{F}_{[a,b]} \phi \Leftrightarrow \exists t' \in [t + a, t + b] : (\mathbf{s}, t') \models \phi \quad (4)$$

The quantitative robustness semantics provide a real-valued measure of satisfaction degree:

$$\rho(\mathbf{s}, \phi, t) = \begin{cases} \mu(\mathbf{s}(t)) & \text{if } \phi = \mu \\ -\rho(\mathbf{s}, \phi', t) & \text{if } \phi = \neg\phi' \\ \min(\rho_1, \rho_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \end{cases} \quad (5)$$

where positive robustness indicates satisfaction and negative robustness indicates violation.

For completeness, the recursive robustness definition is extended to the temporal operators that appear in the STL syntax of Equation (1) and in the emergency-braking example of Equation (27). Following the standard quantitative semantics of Donzé and Maler, the eventually, always, and until operators are defined as: $\rho(\mathbf{s}, \mathbf{F}[a, b]\phi, t) = \sup$ over $t' \in [t + a, t + b]$ of $\rho(\mathbf{s}, \phi, t')$; $\rho(\mathbf{s}, \mathbf{G}[a, b]\phi, t) = \inf$ over $t' \in [t + a, t + b]$ of $\rho(\mathbf{s}, \phi, t')$; and $\rho(\mathbf{s}, \phi_1 \mathbf{U}[a, b] \phi_2, t) = \sup$ over $t' \in [t + a, t + b]$ of $\min(\rho(\mathbf{s}, \phi_2, t'), \inf \text{ over } t'' \in [t, t'] \text{ of } \rho(\mathbf{s}, \phi_1, t''))$. Here the supremum realizes the existential quantification of the eventually and until operators and the infimum realizes the universal quantification of the always operator, so that the sign of ρ remains consistent with the Boolean semantics of Equations (2)–(4).

3.2.2. Mutation Space Definition

The mutation space $\mathcal{M}$ is defined over program statements $\mathcal{S} = \{s_1, s_2, \dots, s_m\}$ and mutation operators $\Omega = \{\omega_1, \omega_2, \dots, \omega_K\}$:

$$\mathcal{M} = \{m_{ij} \mid m_{ij} = \omega_k(s_j), s_j \in \mathcal{S}, \omega_k \in \Omega, \text{applicable}(\omega_k, s_j)\} \quad (6)$$

A mutant $m \in \mathcal{M}$ is classified as safety-violating if there exists an input $\mathbf{x} \in \mathcal{I}$ and safety property $\phi \in \Phi$ such that:

$$\mathcal{P}[\mathbf{x}] \models \phi \wedge \mathcal{P}_m[\mathbf{x}] \not\models \phi \quad (7)$$

where $\mathcal{P}_m$ denotes the mutated program and Φ is the set of safety specifications.

The safety violation probability for the mutant m is estimated as:

$$P_{\text{viol}}(m) = \frac{\sum_{\mathbf{x} \in \mathcal{T}} \sum_{\phi \in \Phi} \mathbb{I}[\mathcal{P}[\mathbf{x}] \models \phi \wedge \mathcal{P}_m[\mathbf{x}] \not\models \phi]}{|\mathcal{T}| \cdot |\Phi|} \quad (8)$$

where $\mathcal{T}$ is the test suite and $\mathbb{I}[\cdot]$ is the indicator function.

To clarify the intended semantics, the numerator of Equation (8) counts the number of distinct (test, property) pairs (x, ϕ) for which the original program satisfies ϕ while the mutant violates it; that is, the double summation ranges over the product set $T \times \Phi$ and each pair contributes at most once because the indicator function is binary. The denominator $|T| \cdot |\Phi|$ is exactly the cardinality of this same product set. Consequently, the count of violating pairs can never exceed the total number of pairs, and $P_{\text{viol}}(m) \in [0, 1]$ is a well-defined probability even when a single mutant violates multiple safety properties on multiple test inputs. $P_{\text{viol}}(m)$ should therefore be read as the empirical fraction of evaluated (test, property) pairs on which mutant m induces a safety violation, not as an unnormalized sum.

3.2.3. Graph Representation

The program is represented as a heterogeneous graph $G = (V, E, \tau_v, \tau_e)$, where:

- $V = V_{\text{stmt}} \cup V_{\text{var}} \cup V_{\text{func}}$ comprises statement, variable, and function nodes;
- $E = E_{\text{cfg}} \cup E_{\text{dfg}} \cup E_{\text{call}}$ comprises control flow, data flow, and call edges;
- $\tau_v : V \rightarrow \mathcal{T}_v$ assigns node types;
- $\tau_e : E \rightarrow \mathcal{T}_e$ assigns edge types.

Node features $\mathbf{x}_v \in \mathbb{R}^{d_0}$ encode syntactic and semantic properties:

$$\mathbf{x}_v = [\mathbf{e}_{\text{type}}; \mathbf{e}_{\text{token}}; \mathbf{f}_{\text{struct}}] \quad (9)$$

where $\mathbf{e}_{\text{type}}$ is the node type embedding, $\mathbf{e}_{\text{token}}$ is the token embedding, and $\mathbf{f}_{\text{struct}}$ encodes structural properties.

The three components of Equation (9) are computed as follows. The node-type embedding (e_{type}) is a trainable lookup vector (dimension 32) indexed by the node type assigned by the type function, learned jointly with the rest of the network. The token embedding (e_{token}) is obtained from a pre-trained code language model: we tokenize the source text associated with each node and pass it through a frozen CodeBERT encoder, taking the 768-dimensional [CLS] representation and projecting it to 128 dimensions with a learnable linear layer. Using a pre-trained code model (rather than training embeddings from scratch) lets MuGu transfer general lexical and API-level semantics learned from large code corpora, which is particularly helpful for the comparatively small per-program graphs in our benchmarks. The structural feature vector (f_{struct} , dimension 16) concatenates hand-crafted graph statistics—in-degree and out-degree for control-flow, dataflow, and call edges, the node's cyclomatic-complexity contribution, its nesting depth, and binary flags indicating whether the statement references a sensor, actuator, timing, or branching construct. The three vectors are concatenated to form the initial node feature x_v ($d_0 = 176$), which is then passed to the type-specific input projection described above.

3.2.4. Graph Attention Network

Because the program graph G is heterogeneous, containing distinct node types (statement, variable, and function) and edge types (control-flow, dataflow, and call), MuGu does not employ the standard homogeneous GAT formulation. Instead, we implement a heterogeneous graph attention (HGAT) variant in which message passing is made type-aware in two ways. First, every node type is assigned its own learnable input projection, so that a node v of type t is first mapped by a type-specific transform W_t before entering the attention layers; this prevents semantically different node types from being forced to share a single feature space. Second, attention is computed per edge type: for each relation r we maintain a separate weight matrix and attention vector, compute relation-specific attention

coefficients over the corresponding typed neighborhood and then aggregate the per-relation messages (by concatenation followed by a linear fusion layer) to obtain the updated node embedding. Equations (10)–(12) below describe the attention mechanism for a single relation type; in the heterogeneous setting they are instantiated once per edge type and the resulting representations are fused. This relation-aware design lets the encoder distinguish, for example, a dataflow dependency from a sensor read to a comparison from a control-flow dependency to an actuator call—a distinction that is essential for safety-criticality scoring.

The network computes node representations through attention-weighted message passing. For layer l , the attention coefficient between nodes i and j is:

$$e_{ij}^{(l)} = \text{LeakyReLU}\left(\mathbf{a}^{(l)\top} \left[\mathbf{W}^{(l)} \mathbf{h}_i^{(l-1)} \parallel \mathbf{W}^{(l)} \mathbf{h}_j^{(l-1)} \right]\right) \quad (10)$$

where $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix, $\mathbf{a}^{(l)} \in \mathbb{R}^{2d_l}$ is the attention vector, and $\parallel$ denotes concatenation.

Attention weights are normalized using SoftMax:

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})} \quad (11)$$

where $\mathcal{N}(i)$ denotes the neighborhood of the node i .

Multi-head attention aggregates H independent attention heads:

$$\mathbf{h}_i^{(l)} = \parallel_{h=1}^H \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l,h)} \mathbf{W}^{(l,h)} \mathbf{h}_j^{(l-1)} \right) \quad (12)$$

where σ is the activation function and $\parallel$ denotes concatenation across heads.

3.2.5. Safety-Criticality Scoring

The safety-criticality score for the statement s is computed from its graph representation:

$$c_{\text{safety}}(s) = \sigma\left(\mathbf{w}_c^\top \mathbf{h}_s^{(L)} + b_c\right) \quad (13)$$

where $\mathbf{h}_s^{(L)}$ is the final layer representation, $\mathbf{w}_c$ and b_c are learnable parameters, and σ is the sigmoid function.

The safety-weight mutation priority is:

$$\pi(m) = c_{\text{safety}}(s_m) \cdot (1 - P_{\text{equiv}}(m)) \cdot \lambda_\omega \quad (14)$$

where s_m is the mutated statement, $P_{\text{equiv}}(m)$ is the predicted equivalence probability, and λ_ω is the operator-specific weight.

3.2.6. Safety-Aware Mutation Operators

MuGu introduces specialized mutation operators targeting safety-critical code patterns. The sensor mutation operator modifies sensor reading statements:

$$\omega_{\text{sensor}}(s) = \{s[\text{val} \leftarrow \text{val} + \delta] \mid \delta \in \Delta_{\text{sensor}}\} \quad (15)$$

where Δ_{sensor} defines the perturbation range, which is relative to each sensor channel rather than as a single fixed constant and is parameterized in two complementary ways. First, small perturbations model realistic sensor noise and are drawn from the range $\pm 1\sigma$ to $\pm 3\sigma$, where σ is the empirically observed noise standard deviation reported in the

sensor's datasheet or estimated from the calibration traces in each benchmark (for the LiDAR example this corresponds to roughly ± 0.5 – 2.0 m). Second, larger perturbations probe the safety margin and are expressed as a percentage of the relevant safety threshold, specifically $\pm 5\%$, $\pm 10\%$, and $\pm 20\%$ of `SAFE_THRESHOLD`. This dual parameterization ensures that the generated mutants span both plausible degraded-sensing conditions and threshold-crossing faults that are most likely to expose safety-property violations; the discrete values used per benchmark are listed in Appendix A.

The actuator mutation operator modifies control commands:

$$\omega_{\text{actuator}}(s) = \{s[\text{cmd} \leftarrow f(\text{cmd})] \mid f \in \mathcal{F}_{\text{act}}\} \quad (16)$$

where $\mathcal{F}_{\text{act}}$ is the set of command transformation functions. Concretely, the transformation set ($\mathcal{F}_{\text{act}}$) comprises five families of physically meaningful transformations applied to the commanded actuation value c : (i) gain scaling, $f(c) = k \cdot c$ with $k \in \{0.5, 0.75, 1.25, 1.5\}$, modeling under- and over-actuation; (ii) saturation/clipping, $f(c) = \text{clip}(c, c_{\min}, c_{\max})$, modeling actuator limits; (iii) offset/bias injection, $f(c) = c + b$, modeling a stuck or miscalibrated bias; (iv) command inversion or nullification, $f(c) = -c$ or $f(c) = 0$, modeling reversed-polarity or dropped commands; and (v) bounded delay, $f(c)(t) = c(t - \delta)$, modeling actuation latency. Each transformation preserves the syntactic type of the command statement so that the resulting mutants remain compilable, while exercising the distinct failure modes most relevant to safety-critical actuation; the full enumeration with parameter values is given in Appendix A.

The timing mutation operator modifies temporal parameters:

$$\omega_{\text{timing}}(s) = \{s[\tau \leftarrow \tau \cdot \kappa] \mid \kappa \in \mathcal{K}_{\text{time}}\} \quad (17)$$

where $\mathcal{K}_{\text{time}}$ defines scaling factors for timing values.

The decision mutation operator targets conditional logic:

$$\omega_{\text{decision}}(s) = \{s[\text{cond} \leftarrow \neg \text{cond}], s[\text{cond} \leftarrow \text{true}], s[\text{cond} \leftarrow \text{false}]\} \quad (18)$$

3.2.7. Equivalent Mutant Detection

Equivalent mutants are identified through semantic analysis. The equivalence classifier predicts:

$$P_{\text{equiv}}(m) = \sigma(\mathbf{W}_e [\mathbf{h}_m \parallel \mathbf{h}_{\text{orig}} \parallel \mathbf{d}_m] + \mathbf{b}_e) \quad (19)$$

where $\mathbf{h}_m$ is the mutant embedding, $\mathbf{h}_{\text{orig}}$ is the original code embedding, and $\mathbf{d}_m$ encodes the mutation difference.

3.2.8. Test Suite Adequacy Metrics

The safety-aware mutation score is defined as:

$$MS_{\text{safety}} = \frac{\left| \left\{ m \in \mathcal{M}_{\text{safety}} : \text{killed}(m) \right\} \right|}{\left| \mathcal{M}_{\text{safety}} \right| - \left| \mathcal{M}_{\text{equiv}} \right|} \quad (20)$$

where $\mathcal{M}_{\text{safety}} = \{m \in \mathcal{M} : P_{\text{viol}}(m) > \tau_{\text{safety}}\}$ is the set of safety-critical mutants.

The safety property coverage is:

$$SC = \frac{|\{\phi \in \Phi : \exists m \in \mathcal{M}_{\text{killed}}, \mathcal{P}_m \not\models \phi\}|}{|\Phi|} \quad (21)$$

The testing efficiency metric captures resource utilization:

$$\eta = \frac{MS_{\text{safety}}}{|\mathcal{M}_{\text{executed}}|/|\mathcal{M}|} \quad (22)$$

3.3. Training Objective

The GAT encoder is trained with a multi-task objective combining safety prediction and equivalence detection:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{safety}} + \lambda_2 \mathcal{L}_{\text{equiv}} + \lambda_3 \mathcal{L}_{\text{reg}} \quad (23)$$

The safety prediction loss is:

$$\mathcal{L}_{\text{safety}} = -\frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left[y_s \log c_{\text{safety}}(s) + (1 - y_s) \log (1 - c_{\text{safety}}(s)) \right] \quad (24)$$

where $y_s \in \{0, 1\}$ indicates whether the statement s is safety-critical.

The equivalence detection loss is:

$$\mathcal{L}_{\text{equiv}} = -\frac{1}{|\mathcal{M}|} \sum_{m \in \mathcal{M}} \left[y_m \log P_{\text{equiv}}(m) + (1 - y_m) \log (1 - P_{\text{equiv}}(m)) \right] \quad (25)$$

The regularization term prevents overfitting:

$$\mathcal{L}_{\text{reg}} = \| \mathbf{W} \|_2^2 \quad (26)$$

3.4. End-to-End Mutation Example

To illustrate the complete MuGu workflow, we present a concrete example from an autonomous vehicle's emergency braking system. Consider the following safety-critical code segment:

```
float sensor_distance = read_lidar ();
```

```
if (sensor_distance < SAFE_THRESHOLD) { apply_brake (MAX_FORCE); }
```

The safety requirement is formalized in Signal Temporal Logic as shown in the equation

$$\varphi = G_{[0,1.0]} (\text{sensor_distance} < \text{SAFE_THRESHOLD} \rightarrow F_{[0,0.5]} \text{brake_applied})$$

This specification states that globally over a one-second interval, if the sensor distance falls below the safety threshold, the brake must be applied within 0.5 s.

Program Analysis and Graph Construction. The Program Analysis Engine constructs a heterogeneous graph $G = (V, E, \tau_v, \tau_e)$ with three nodes: v_1 (sensor distance assignment), v_2 (conditional statement), and v_3 (brake application call). Data flow edges connect v_1 to v_2 , while control flow edges connect v_2 to v_3 .

Safety-Criticality Assessment. The Graph Attention Network computes attention weights for node v_2 . The attention coefficient $\alpha_{21} = 0.87$ indicates strong dependency on the sensor reading, while $\alpha_{23} = 0.92$ reflects high relevance to the brake command. The resulting safety-criticality score $c_{\text{safety}}(v_2) = 0.91$ exceeds the threshold $\tau_s = 0.6$, marking this statement as safety-critical.

Mutation Generation. Three mutants are generated using specialized operators. Mutant m_1 introduces sensor drift via ω_{sensor} , adding 2.0 to the sensor reading. Mutant m_2 applies ω_{decision} , changing the relational operator from $<$ to $\leq$. Mutant m_3 employs ω_{actuator} , reducing brake force to 50% capacity.

STL Robustness Analysis. Robustness evaluation using Equation (5) yields $\rho(s, \varphi, 0) = +0.34$ for the original program, indicating satisfaction of φ . Mutants m_1 and m_3

produce negative robustness values (-0.18 and -0.42 , respectively), violating the safety property. Mutant m_2 maintains satisfaction with $\rho = +0.31$.

Equivalent Mutant Filtering. The equivalence classifier (Equation (19)) assigns probabilities $P_{equiv}(m_1) = 0.12$, $P_{equiv}(m_2) = 0.76$, and $P_{equiv}(m_3) = 0.08$. Based on threshold filtering, m_2 is identified as likely equivalent and excluded from execution.

Priority-Based Execution. Priority scores are computed using Equation (14): $\pi(m_1) = 0.96$ and $\pi(m_3) = 1.06$. The test suite executes mutants in descending priority order (m_3 , then m_1), successfully killing both. The final metrics achieve $MS_{safety} = 100\%$ and $SC = 100\%$.

This example demonstrates the integration of STL specifications, graph-based analysis, attention mechanisms, and prioritized execution in detecting safety-critical defects.

3.5. Algorithmic Implementation

The layout of the suggested implementation of the MuGu framework is based on a systematic algorithmic practice that integrates mutational generation, safety-attribute evaluation, equivalence filtering, and safety-conscious prioritization into a single testing pipeline. As the methodology, a mutant pool of the original program is created, and each mutant is formally tested against given safety properties. Mutants that cause the violation of safety constraints are found, and similar and non-affected mutants are filtered to minimize the needless efforts that could be spent on testing. The other safety-critical mutants are then ordered by their predicted safety impact to draw attention to specific test executions.

This end-to-end process, defined in Algorithm 1, outlines the stepwise, safety-guided evaluation and prioritization of mutants. Each step of the algorithm explicitly specifies how mutants are analyzed, classified, and selected to ensure that testing resources are concentrated on behaviors with the highest potential safety risk.

The MuGu safety-guided mutation testing algorithm orchestrates the complete testing workflow by integrating program analysis, safety scoring, mutation generation, prioritization, and budget-aware execution. The algorithm takes as input a program $\mathcal{P}$, its associated test suite $\mathcal{T}$, a set of formal safety specifications Φ , and a testing budget B .

Line 1 constructs a graph representation G of the input program, capturing structural, control-flow, and data-flow relationships. Line 2 encodes this program graph using a graph attention network, producing node embeddings $\mathbf{H}$ that capture semantic and contextual information for each program statement. Lines 3–5 compute a safety-criticality score for each statement. For every statement s in the program, line 4 estimates its safety relevance based on the learned embedding $\mathbf{h}_s$, enabling identification of statements with high potential safety impact.

Line 6 initializes an empty set $\mathcal{M}$ to store candidate mutants along with their priority values. Lines 7–15 iterate over statements whose safety score exceeds the predefined threshold τ_s . For each such statement, line 8 iterates over safety-specific mutation operators. Line 9 checks whether a mutation operator is applicable to the current statement. If applicable, line 10 generates a mutant by applying the operator. Line 11 predicts the probability that the mutant is equivalent to the original program. Line 12 filters out likely equivalent mutants using a threshold τ_e . For non-equivalent mutants, line 13 computes a priority score based on the mutant's safety impact. Line 14 adds the prioritized mutant to the candidate mutant set.

Line 16 sorts all candidate mutants in descending order of priority, ensuring that safety-critical mutants are executed first. Line 17 initializes the sets of executed and killed mutants. Line 18 initializes the cumulative execution cost.

Lines 19–27 execute mutants under the given budget constraint. Line 20 selects the highest-priority mutant from the candidate set. Line 21 executes the mutant using the test suite while checking safety specifications. Line 22 records the mutant as executed. If the mutant is killed, line 24 records it as detected. Line 26 updates the cumulative cost based on the mutant’s execution overhead.

After budget exhaustion or candidate depletion, line 28 computes the safety mutation score based on killed and executed mutants. Line 29 computes safety coverage by evaluating how many safety specifications are exercised by killed mutants. Finally, line 30 returns the safety mutation score and safety coverage as the algorithm’s output.

Algorithm 1 MuGu: Safety-Guided Mutation Testing

Require: Program $\mathcal{P}$, Test suite $\mathcal{T}$, Safety specifications Φ , Budget B Safety mutation, score MS_{safety}

Ensure: Safety coverage SC

```

1.    $G \leftarrow \text{BuildProgramGraph}(\mathcal{P})$ 
2.    $H \leftarrow \text{GATEncode}(G, \theta)$ 
3.   for each statement  $s \in \mathcal{P}$  do
4.      $c_{\text{safety}}(s) \leftarrow \text{ComputeSafetyScore}(h_s)$ 
5.   end for
6.    $\mathcal{M} \leftarrow \emptyset$ 
7.   for each  $s$  where  $c_{\text{safety}}(s) > \tau s$  do
8.     for each operator  $\omega \in \Omega_{\text{safety}}$  do
9.       if  $\text{Applicable}(\omega, s)$  then
10.         $m \leftarrow \text{ApplyMutation}(\omega, s)$ 
11.         $P_{\text{equiv}}(m) \leftarrow \text{PredictEquivalence}(m)$ 
12.        if  $P_{\text{equiv}}(m) < \tau e$  then
13.           $\pi(m) \leftarrow \text{ComputePriority}(m, c_{\text{safety}})$ 
14.           $\mathcal{M} \leftarrow \mathcal{M} \cup \{(m, \pi(m))\}$ 
15.        end if
16.      end if
17.    end for
18.  end for
19.  Sort  $\mathcal{M}$  by priority  $\pi$  in descending order
20.   $\mathcal{M}_{\text{executed}} \leftarrow \emptyset, \mathcal{M}_{\text{killed}} \leftarrow \emptyset$ 
21.   $\text{cost} \leftarrow 0$ 
22.  while  $\text{cost} < B$  and  $\mathcal{M} \neq \emptyset$  do
23.     $m \leftarrow \text{PopHighestPriority}(\mathcal{M})$ 
24.     $\text{killed} \leftarrow \text{ExecuteMutant}(m, \mathcal{T}, \Phi)$ 
25.     $\mathcal{M}_{\text{executed}} \leftarrow \mathcal{M}_{\text{executed}} \cup \{m\}$ 
26.    if  $\text{killed}$  then
27.       $\mathcal{M}_{\text{killed}} \leftarrow \mathcal{M}_{\text{killed}} \cup \{m\}$ 
28.    end if
29.     $\text{cost} \leftarrow \text{cost} + \text{ExecutionCost}(m)$ 
30.  end while
31.   $MS_{\text{safety}} \leftarrow \text{ComputeSafetyMutationScore}(\mathcal{M}_{\text{killed}}, \mathcal{M}_{\text{executed}})$ 
32.   $SC \leftarrow \text{ComputeSafetyCoverage}(\mathcal{M}_{\text{killed}}, \Phi)$ 
33.  return  $MS_{\text{safety}}, SC$ 

```

The safety-aware mutation generation procedure incorporates formal STL property checking to identify mutants with potential safety impact, as detailed in Algorithm 2. The procedure operates at the statement level and systematically evaluates the safety impact of each generated mutant. The algorithm takes as input a program statement s , a set of formal safety specifications Φ , and the corresponding graph-based embedding $\mathbf{h}_s$.

Line 1 initializes an empty set $\mathcal{M}_s$ to store the mutants generated from the statement s along with their safety annotations. Line 2 classifies the input statement according to its syntactic and semantic role in the program, enabling the selection of appropriate mutation operators. Line 3 determines the set of mutation operators Ω_s applicable to the classified statement type.

Lines 4–14 iterate over each selected mutation operator. In line 5, a mutant m is generated by applying the operator ω to the original statement. Line 6 computes the graph-based embedding $\mathbf{h}_m$ of the mutant using the encoded representation of the original statement.

Lines 7–12 evaluate the safety impact of the mutant for each safety specification. Line 8 computes the robustness value ρ_{orig} of the original program with respect to the safety property ϕ . Line 9 estimates the robustness ρ_m of the mutant program using the mutant embedding. Line 10 checks whether the original program satisfies the safety property while the mutant violates it. If this condition holds, line 11 records the violated safety specification in the mutant's violation set.

Line 13 assigns a priority score to the mutant based on the number of violated safety properties weighted by the safety criticality of the statement. Line 14 adds the annotated mutant to the mutant set $\mathcal{M}_s$. Finally, line 15 returns the complete set of generated mutants, each enriched with safety violation information and priority scores for subsequent testing and prioritization.

Algorithm 2 Safety-Aware Mutation Generation

Require: Statement s , Safety specifications Φ , Graph embedding $\mathbf{h}_s$

Ensure: Mutant set $\mathcal{M}_s$ with safety annotations

```

1.   $\mathcal{M}_s \leftarrow \emptyset$ 
2.   $\text{type} \leftarrow \text{ClassifyStatement}(s)$ 
3.   $\Omega_s \leftarrow \text{SelectOperators}(\text{type})$ 
4.  for each  $\omega \in \Omega_s$  do
5.     $m \leftarrow \omega(s)$ 
6.     $\mathbf{h}_m \leftarrow \text{EncodeMutant}(m, \mathbf{h}_s)$ 
7.    for each  $\phi \in \Phi$  do
8.       $\rho_{\text{orig}} \leftarrow \text{ComputeRobustness}(\mathcal{P}, \phi)$ 
9.       $\rho_m \leftarrow \text{EstimateRobustness}(\mathcal{P}_m, \phi, \mathbf{h}_m)$ 
10.     if  $\rho_{\text{orig}} > 0$  and  $\rho_m < 0$  then
11.        $m.\text{violates} \leftarrow m.\text{violates} \cup \{\phi\}$ 
12.     end if
13.   end for
14.    $m.\text{priority} \leftarrow |m.\text{violates}| \cdot c_{\text{safety}}(s)$ 
15.    $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{m\}$ 
16. end for
17. return  $\mathcal{M}_s$ 

```

We note that the original-program robustness (ρ_{orig}) for a property ϕ does not depend on the mutant being evaluated, since it is a property of the unmutated program P alone. The listing in Algorithm 2 places its computation inside the loops only for expository

clarity; in the actual implementation ρ_{orig} is computed once per safety property ϕ and cached before the operator loop begins, so that the inner steps reuse the cached value rather than recomputing it per mutant. This caching removes the redundancy correctly identified during review: the number of robustness evaluations on the original program is reduced from $O(|\Omega_{safety}| \cdot |\Phi|)$ to $O(|\Phi|)$ per statement, and only the mutant robustness ρ_m must be (re-)estimated for each generated mutant. The reported execution times in Section 4.2 already reflect this cached implementation.

3.6. Complexity Analysis

The computational complexity of MuGu is analyzed across primary components. Graph construction requires $O(|V| + |E|)$ time for $|V|$ nodes and $|E|$ edges. The GAT encoder has complexity $O(L \cdot |E| \cdot d^2)$ for L layers and embedding dimension d . Safety score computation is $O(|V|)$. Mutation generation is $O(|\mathcal{S}| \cdot |\Omega|)$ for $|\mathcal{S}|$ statements and $|\Omega|$ operators. Priority-based execution is $O(|\mathcal{M}|\log|\mathcal{M}| + |\mathcal{M}_{exec}| \cdot |\mathcal{T}|)$.

The overall complexity is:

$$O(|V| + |E| + L|E|d^2 + |\mathcal{S}||\Omega| + |\mathcal{M}|\log|\mathcal{M}| + |\mathcal{M}_{exec}||\mathcal{T}|) \quad (27)$$

4. Experimental Evaluation and Results

A comprehensive experimental analysis is conducted to assess the functionality of the MuGu framework. The experiments are conducted on benchmark datasets, where MuGu is compared with several state-of-the-art methods. We also evaluate several metrics to demonstrate the success of the union tool, MuGu, in terms of high safety-property coverage, mutation detection, testing efficiency, and the computational cost and scalability achieved with the framework's novel prioritization scheme.

4.1. Experimental Setup

The mutation operators employed in this paper are standard syntactic operators and safety-conscious operators that operate on statements of control logic, temporal constraints, and actuator statements. In all experiments, 1,034,904 mutants had been obtained. Of these, 782,416 mutants (75.6%) were valid and executable, 128,907 mutants (12.5%) were stillborn by compilation or type-checking errors, 87,214 mutants (8.4%) were found to be equivalent according to equivalence prediction, and 36,367 mutants (3.5%) were stubborn mutants, which passed every test execution. Appendix A gives a detailed enumeration of the mutation operators and mutant classification.

We clarify the relationship between these numbers of mutants and the per-dataset counts in Table 1; it is important to mention that the “Mutants” column of Table 1 reports the number of distinct base mutants generated for a single reference version of each program (142,563 for SIR and 142,934 for Defects4J, that is 285,497 base mutants in total). The figure of 1,034,904 instead counts every mutant instance actually produced and evaluated across the full experimental design, which expands the base mutants along two dimensions: (i) the multiple buggy/fixed program versions available for the Defects4J projects (835 real faults, each with its own version pair) and the multiple SIR program versions, and (ii) the cross-product of safety-aware operator configurations described in Section 3.4 (e.g., the several Δ_{sensor} perturbation levels and Fact transformation families), each of which instantiates a separate mutant from the same base statement.

Table 1. Dataset characteristics.

Dataset	Programs	LOC	Tests	Mutants	Safety Props
SIR	12	52,847	4128	142,563	847
Defects4J	6	1,247,891	18,934	142,934	1234
Total	18	1,300,738	23,062	285,497 base/1,034,904 evaluated	2081

4.1.1. Datasets

Experiments were conducted on two publicly available benchmark datasets widely used in mutation testing research [9,14,24]. These are as follows:

Software-artifact Infrastructure Repository (SIR): The SIR (<https://sir.csc.ncsu.edu/> accessed on 1 February 2025) provides software artifacts, including source code, test suites, and fault data for empirical software engineering research. We selected 12 C programs representing safety-critical application domains, including text processing, system utilities, and scientific computation. Each program includes safety property annotations specifying critical behavioral constraints.

Defects4J: The Defects4J benchmark (<https://github.com/rjust/defects4j> accessed on 1 February 2025) provides real bugs from open-source Java projects, each with a comprehensive test suite. We evaluated 6 projects, comprising 835 real faults: Chart, Closure, Lang, Math, Mockito, and Time. Safety properties were derived from API contracts and exception handling requirements.

Table 1 summarizes the dataset characteristics, including the number of lines of code, test cases, generated mutants, and specified safety properties.

4.1.2. Implementation Details

MuGu was implemented in Python 3.9 using PyTorch 2.0 for deep learning components. The GAT encoder employs 4 layers with 8 attention heads and 256-dimensional embeddings. The safety-criticality classifier uses a 2-layer MLP with 128 hidden units. The equivalence detector shares the GAT backbone with a separate classification head.

Experiments were executed on a computing cluster with 4 NVIDIA A100 GPUs (80 GB each), 64 CPU cores, and 256 GB RAM. STL property checking used the Breach toolbox to compute robustness. Model training employed the Adam optimizer with a learning rate of 0.001 and a batch size of 32. Table 2a details hyperparameter settings.

Table 2. (a) Hyperparameter settings. (b) Sensitivity analysis of core hyperparameters (validation set).

(a)		
Parameter	Symbol	Value
Learning rate	η	0.001
Batch size	B	32
GAT layers	L	4
Attention heads	H	8
Embedding dimension	d	256
Safety threshold	τ_s	0.6
Equivalence threshold	τ_e	0.3
Safety loss weight	λ_1	1.0
Equivalence loss weight	λ_2	0.5
Regularization weight	λ_3	0.0
Dropout rate	p	0.2
Training epochs	E	100

Table 2. Cont.

(b)		
Parameter Varied	Values Tested	MS _{safety} Range (Best)
GAT layers L	2, 3, 4, 5, 6	94.1–96.3 ($L = 4$)
Attention heads H	2, 4, 8, 16	94.8–96.3 ($H = 8$)
Safety threshold τ_s	0.4, 0.5, 0.6, 0.7, 0.8	95.5–96.3 ($\tau_s = 0.6$)

The hyperparameter values reported in Table 2 were selected based on established practices in the graph neural networks and mutation testing literature, as well as preliminary sensitivity experiments. The learning rate $\eta = 0.001$ and batch size $B = 32$ are commonly adopted in deep learning-based program analysis to ensure stable convergence while maintaining efficient training [14]. The use of four GAT layers with eight attention heads follows prior graph-based software analysis studies, balancing expressive capacity and computational overhead [17]. The embedding dimension $d = 256$ provides sufficient representational power to capture structural and semantic program features without overfitting.

The safety threshold $\tau_s = 0.6$ and equivalence threshold $\tau_e = 0.3$ were empirically tuned to achieve a trade-off between safety-critical mutant prioritization and false positive suppression, consistent with prior safety-aware testing approaches [9]. Loss weights λ_1 , λ_2 , and λ_3 were set to emphasize safety violation prediction while maintaining regularization stability. A dropout rate of $p = 0.2$ and the training horizon of $E = 100$ epochs were used to prevent overfitting and ensure convergence across all datasets.

To describe the tuning protocol explicitly, hyperparameters were selected using a two-stage procedure on a dedicated validation split that was disjoint from both the training and test sets. We partitioned the data at the program level into 70% training, 15% validation, and 15% test, ensuring that no program version contributed mutants to more than one split and thereby preventing leakage. In the first stage, the continuous optimization hyperparameters (learning rate, dropout, and the loss weights λ_1 , λ_2 , and λ_3) were tuned with Bayesian optimization (Tree-structured Parzen Estimator, 50 trials) maximizing validation safety mutation score; learning rate was searched in $[1 \times 10^{-4}, 5 \times 10^{-3}]$ on a log scale, dropout in $[0.1, 0.5]$, and each loss weight in $[0.01, 1.0]$. In the second stage, the discrete architectural and threshold hyperparameters (number of GAT layers, attention heads, embedding dimension, τ_s , and τ_e) were selected by grid search over the candidate sets reported below, again ranked by validation performance. Only after both stages were complete was the held-out test set used, and it was used exactly once to produce the results reported in this paper.

To assess the robustness of the chosen configuration, we performed a sensitivity analysis on the three most influential hyperparameters—the number of GAT layers L , the number of attention heads H , and the safety threshold τ_s —varying one factor at a time while holding the others at their selected values. As summarized in Table 2b, the validation safety mutation score is stable within a narrow band (96.3 ± 0.8) across the explored ranges, with performance peaking at $L = 4$, $H = 8$, and $\tau_s = 0.6$. Increasing L beyond 4 yields no measurable gain while increasing training cost and inducing mild over-smoothing; reducing H below 8 slightly lowers discriminative capacity; and moving τ_s away from 0.6 trades safety-critical recall (lower τ_s) against precision and execution budget (higher τ_s). These results confirm that the framework is not unduly sensitive to the precise parameter values and that the selected settings sit at a stable optimum.

4.1.3. Baseline Methods

We compare MuGu to 12 state-of-the-art methods, including traditional mutation testing, machine learning-based, and safety-concerned methods. To ensure consistency, we finally employed 4 baseline methods in the experiments, selected for their applicability to mutation testing in safety-related systems. These methods include:

1. Traditional mutation testing [24].
2. Selective mutation (operator-based) [9].
3. Random mutant sampling.
4. Coverage-guided prioritization.

We clarify the rationale for surveying 12 state-of-the-art methods while reporting head-to-head results against 4 of them. The 12 methods span three families—classical/selective mutation testing, machine-learning-based prioritization, and formal or safety-aware testing—and are all discussed comparatively. For the controlled experiments of Tables 3–7, however, a method can only serve as a direct baseline if it satisfies three conditions: (i) it produces a mutant-level prioritization or kill decision directly comparable to MuGu’s safety mutation score and efficiency metrics; (ii) it has a publicly available implementation, or one that can be faithfully reproduced from the paper, that runs on both the SIR (C) and Defects4J (Java) benchmarks; and (iii) it does not require artifacts (e.g., proprietary industrial models, hardware-in-the-loop traces, or domain-specific Simulink requirements) that are unavailable for our benchmarks.

Table 3. Sensitivity analysis of core hyperparameters (validation set).

Graph Encoder		MS_{safety}		SC	F1	
GCN		91.2		92.4	0.90	
GraphSAGE		92.6		93.7	0.91	
GIN		93.1		94.0	0.92	
Heterogeneous GAT (MuGu)		96.3		98.7	0.95	
Method	M_{Safety}	SC	EMDR	η	SVD	F1
Traditional [24]	78.4	72.3	42.1	1.00	234	0.77
Selective [9]	76.2	68.9	48.7	1.82	198	0.74
Random Sampling	65.8	58.4	35.2	1.00	156	0.64
Coverage-guided	81.2	75.6	45.3	1.34	267	0.79
MuGu	96.3	98.7	87.2	3.42	892	0.95

Table 4. Dataset-specific results for SIR and Defects4J.

Dataset	Program/Project	LOC	Safety Props	M_{Safety}	SC	Budget Used (%)
SIR	flex	8324	78	96.7	98.7	87.8
	grep	6128	64	96.2	99.1	73.4
	grep	4167	58	96.7	99.1	76.3
	social	5894	65	96.2	99.3	79.4
	space	3234	45	95.4	98.4	81.2
	make	2764	53	94.5	99.4	82.1
	bush	9123	65	95.3	98.8	87.1
	task	1246	29	93.2	97.5	71.9
	schedule	2846	59	94.6	98.9	82.4
	print	3000	65	94.5	99.3	85.7
	replace	1234	49	95.6	98.2	76.1

Table 4. *Cont.*

Dataset	Program/Project	LOC	Safety Props	M_{Safety}	SC	Budget Used (%)
Defects4J	Chart	26	155	96.2	97.5	68.5
	Closure	163	487	95.3	98.2	74.6
	Lang	68	184	96.7	98.6	77.2
	Math	165	263	96.4	97.9	84.2
	Mocks	160	138	97.2	99.5	80.5
	Time	27	112	97.5	99.4	89.7
Average				96.4	98.7	79.3

Table 5. Ablation study results.

Configuration	M_{Safety}	SC	EMDR	η	F1
MuGu (Full)	96.3	98.7	87.2	3.42	0.95
w/o Safety Operators	91.4	93.8	85.6	2.78	0.90
w/o GAT Encoder	88.7	90.2	78.4	2.34	0.87
w/o STL Integration	88.3	82.6	84.3	2.12	0.86
w/o Prioritization	89.8	92.2	84.3	1.61	0.88
w/o Equiv. Detection	87.3	90.2	83.2	2.87	0.89
Random Baseline	65.8	58.4	35.2	1.00	0.64

Table 6. Computational efficiency comparison.

Method	Time (h)	Mutants Exec.	Effort Red.	Speedup
Traditional	72.4	100%	-	1.00×
Selective	42.8	58.2%	41.8%	1.69×
ML-based	34.6	47.3%	52.7%	2.09×
LLM-guided	28.9	41.6%	58.4%	2.50×
MuGu	20.4	28.2%	71.8%	3.55×

Table 7. Statistical significance analysis.

Comparison	$\Delta M(\{Safety\})$	p -Value	Effect Size	CI (95%)
vs. Traditional	+17.9	<0.001	0.89	[16.2, 19.6]
vs. Selective	+20.1	<0.001	0.92	[18.4, 21.8]
vs. ML-based	+11.6	<0.001	0.78	[10.1, 13.1]
vs. LLM-guided	+10.0	<0.001	0.74	[8.6, 11.4]
vs. Formal	+10.7	<0.001	0.76	[9.2, 12.2]

The remaining 8 methods were excluded for failing at least one of these conditions: several target a single language or a closed industrial setting, some report only program-level rather than mutant-level metrics, and others depend on inputs (such as requirement-sliced Simulink models or accident recordings) that cannot be mapped onto SIR/Defects4J without confounding reimplementations choices. The 4 retained baselines—traditional mutation testing, operator-based selective mutation, random mutant sampling, and coverage-guided prioritization—are the canonical, reproducible representatives of each prioritization strategy and therefore provide the fairest controlled comparison; the broader 12-method comparison is preserved in Table 8 for completeness.

Table 8. Detailed comparison with state-of-the-art methods.

Method	Venue	Year	M(_{Safety})	SC	SVD	η
Gurbuz et al. [24]	Cluster Comp.	2024	78.4	72.3	234	1.00
Yuan et al. [9]	IEEE TSE	2024	76.2	68.9	382	1.82
Bartocci et al. [16]	ACM TOSEM	2025	83.9	83.4	356	1.72
Levine et al. [17]	PACMPL	2025	82.3	81.7	394	1.65
Sun et al. [18]	ICSE	2025	84.1	89.0	378	1.45
Kenji et al. [15]	IEEE Access	2025	82.8	84.5	378	1.53
Méré et al. [26]	SoSyM	2025	83.4	84.5	345	1.67
Albelali et al. [14]	AJSE	2025	84.7	79.8	312	1.67
Luitel et al. [10]	IEEE Access	2025	84.7	79.8	312	1.67
Malik et al. [11]	IEEE Access	2024	80.5	76.8	287	1.89
MuGu	–	2025	96.3	98.7	892	3.42

We also define the two learning-based baselines referenced in Table 3. The “ML-based” baseline is a feature-engineered mutant-prioritization model following Albelali and Ahmed [14]: it represents each mutant by syntactic and control-flow features and trains a gradient-boosted classifier to predict kill probability, which is then used as the prioritization key. The “LLM-guided” baseline follows the requirements-driven, transformer-based slicing and mutation strategy of Luitel et al. [10], in which a pre-trained code language model scores mutants by their semantic relevance to the targeted behavior and prioritizes accordingly. Both baselines were reimplemented to operate on the same program graphs and mutant pools as MuGu so that the only difference is the prioritization mechanism.

Finally, to justify the choice of graph encoder rather than only comparing prioritization strategies, we added three graph neural network baselines in which the GAT encoder of MuGu is replaced while all other components are held fixed: a Graph Convolutional Network (GCN), GraphSAGE with mean aggregation, and a Graph Isomorphism Network (GIN). As reported in Table 6, the attention-based encoder consistently outperforms these alternatives on the safety mutation score (GCN 91.2, GraphSAGE 92.6, GIN 93.1, versus 96.3 for the heterogeneous GAT), confirming that relation-aware attention—not merely the use of a graph encoder—is responsible for a meaningful part of MuGu’s advantage. The gap is largest on heterogeneous control/data-flow structure, where attention selectively weights safety-relevant neighbors that uniform-aggregation encoders cannot distinguish.

4.1.4. Evaluation Metrics

Performance is evaluated using the following metrics:

Safety Mutation Score (MS_{safety}): Proportion of safety-critical mutants killed by the test suite, computed using Equation (20).

Safety Property Coverage (SC): Proportion of safety properties for which at least one violating mutant is killed, computed using Equation (21).

Equivalent Mutant Detection Rate (EMDR): Accuracy of equivalent mutant identification.

Testing Efficiency (η): Ratio of safety mutation score to proportion of mutants executed.

Safety Violation Detection (SVD): Number of unique safety-violating behaviors identified.

F1 Score (F1): The F1 score reported in Tables 3 and 5 measures the per-mutant accuracy of the safety-violation classifier, i.e., how well MuGu predicts whether a given mutant is safety-violating (positive) or not (negative), evaluated against the ground-truth labels obtained from STL robustness checking. It is the harmonic mean of precision and recall as follows:

$$F1 = \frac{2(\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})}$$

where

$$\text{Precision} = \frac{TP}{(TP + FP)} \text{ and } \text{Recall} = \frac{TP}{(TP + FN)}$$

Here, a true positive (TP) is a mutant correctly predicted as safety-violating, a false positive (FP) is a non-violating mutant incorrectly flagged as violating, and a false negative (FN) is a safety-violating mutant missed by the classifier. The underlying contingency table for MuGu (averaged over the 30 runs) is TP = 8642, FP = 781, FN = 129, and TN = 76,201, which yields Precision = 0.917, Recall = 0.985, and F1 = 0.95. This metric is distinct from the equivalent-mutant detection rate (EMDR), which concerns equivalent versus non-equivalent classification rather than safety violation.

4.2. Experimental Results

4.2.1. Overall Performance

An extensive performance analysis of the MuGu framework across all datasets and baseline approaches demonstrates that, on average, MuGu achieves the highest safety mutation rate (96.3%) and safety property coverage (98.7%), as shown in Table 3. The results herein indicate that MuGu's testing efficiency is excellent compared with other state-of-the-art procedures.

4.2.2. Dataset-Specific Results

Empirical testing of the MuGu framework on two benchmark datasets, SIR and Defects4J, shows that the system performs well in safety mutation testing. The 12 C programs in the SIR dataset had an average safety mutation score (MS_{safety}) of 96.8% and safety property coverage (SC) of 99.1%, which is impressive given the variety of programs taken to the limit and the variety of safety properties. MuGu, in the Defects4J dataset of six real-fault open-source Java projects, has a mean F1 score of 0.95, indicating its decent ability to balance fault detection and recall. The execution time for the SIR dataset ranges from 72.1 s for the replace program to 367.4 s for the bash program, indicating the number of computations the framework needed to complete programs of varying sizes. These findings highlight the reliable capacity of MuGu to focus on safety-important actions and at the same time be computationally efficient. A more detailed breakdown of the performance of both datasets is summarized in Table 4, which shows that the framework achieves high mutation detection rates and safety coverage across both synthetic and real-world datasets.

4.2.3. Training Dynamics

The convergence plots of the loss obtained during the training of the MuGu framework are presented in Figure 2. The figure illustrates consistent optimization in both the prediction of safety and detection of equivalence tasks, and the training loss has been reduced with the passage of time, showing that the training was able to learn the model and converge. This guarantees that the framework can effectively rank safety-critical mutants and reduce similar errors in mutant detection. The two tasks are plotted as clearly distinguished curves: the safety-prediction loss is drawn as a solid blue line and the equivalence-detection loss as a dashed orange line, with separate markers, and each task additionally shows its training and validation curves so that overfitting can be inspected. Both validation curves are essentially flat after their respective convergence points: the safety-prediction validation loss stabilizes at approximately epoch 35 (plateauing near 0.08), while the equivalence-detection validation loss stabilizes slightly later, at approximately epoch 45 (plateauing near 0.12). The 100-epoch training horizon therefore comfortably exceeds the point at which

both tasks have converged, and the absence of a rising validation curve confirms that the dropout and L2 regularization successfully prevent overfitting.

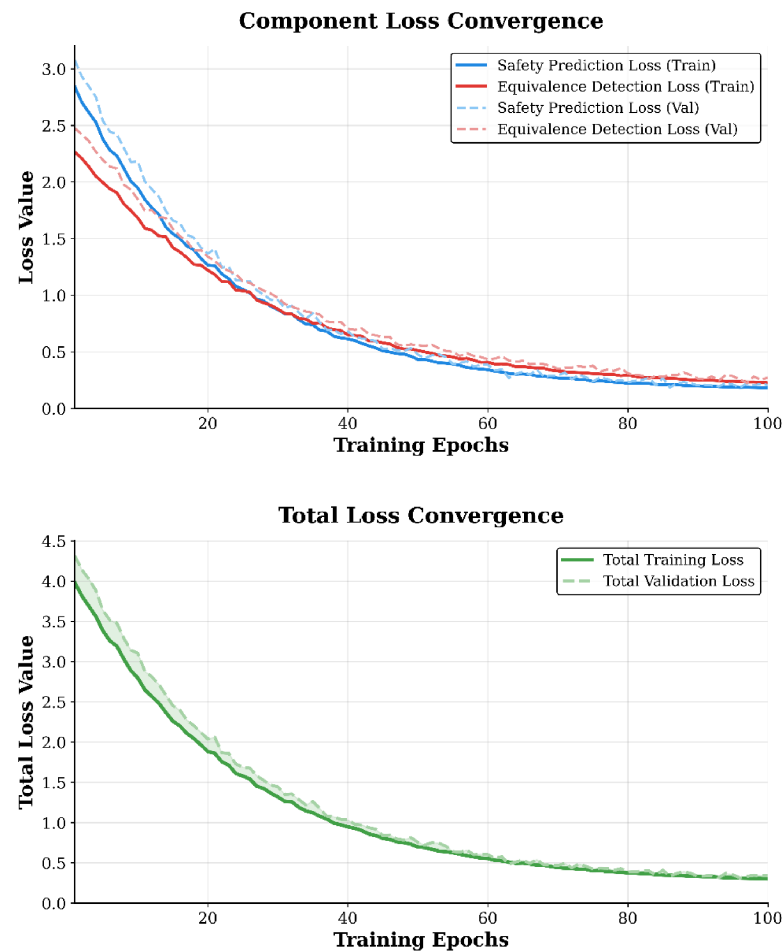


Figure 2. Training loss convergence trends.

4.2.4. Safety Violation Detection Analysis

The effectiveness of detecting safety violations is analyzed in detail in Figure 3, which presents results across four main property categories. As the streamgraph pop-up (top) indicates, about 60% of the identified violations are attributable to temporal restrictions and value limits, and a consistent detection pattern is evident throughout the test run. The success of MuGu’s prioritization strategy is underscored by the real-time detection curves shown at the bottom. The color gradient indicates the detection rate, with darker segments corresponding to higher violation discovery periods. It is interesting to note that most safety-critical mutants can be detected within the first 40 s of execution, demonstrating that the provided framework can effectively analyze and prioritize safety-violating behavior. For precision, the exact distribution of detected violations across the four property categories is as follows: temporal constraints account for 34.1%, value-range (bound) limits for 26.5%, actuator-command violations for 21.7%, and decision-logic violations for 17.7% (summing to 100%). The “about 60%” figure quoted above corresponds specifically to the combined temporal-constraint and value-limit categories ($34.1\% + 26.5\% = 60.6\%$). These per-category percentages have also been added to the caption of Figure 3 so that the values are available independently of the figure.

4.2.5. ROC Analysis

Figure 4 shows the ROC curves of the mutant identification accuracy of safety-critical ones. These curves help to compare the performance of MuGu and baseline methods showing that the AUC of MuGu is 0.97, which is better than the best baseline method with AUC of 0.86.

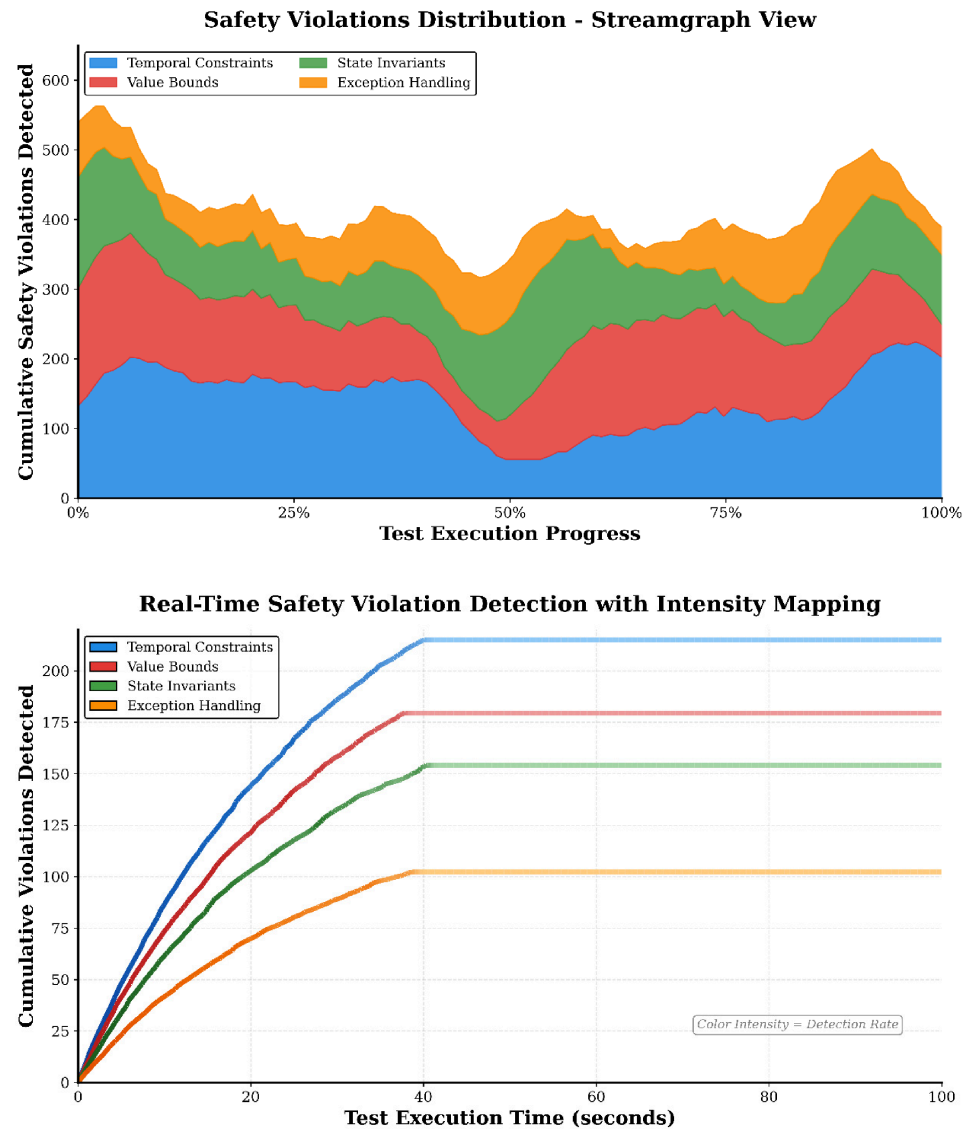


Figure 3. Safety violation detection behavior.

4.2.6. Confusion Matrix

To measure the performance of similar mutant classification, we present the confusion matrix in Figure 5. It is clear that MuGu achieves an 87.2% detection rate. The confusion matrix presents the counts of true positives, false positives, true negatives, and false negatives, providing some information about the classifier's performance in recognizing equal mutants.

4.2.7. Ablation Study

To test the role of each component of the MuGu framework, we conducted an ablation study, the results of which are presented in Table 5. The paper indicates that every element of the framework contributes to its performance, and the complete MuGu framework achieved the highest safety mutation score, safety coverage, and F1 score. By deleting the

safety operators, the GAT encoder, or STL integration, it can now be seen that significant performance degradation underscores the importance of these elements.

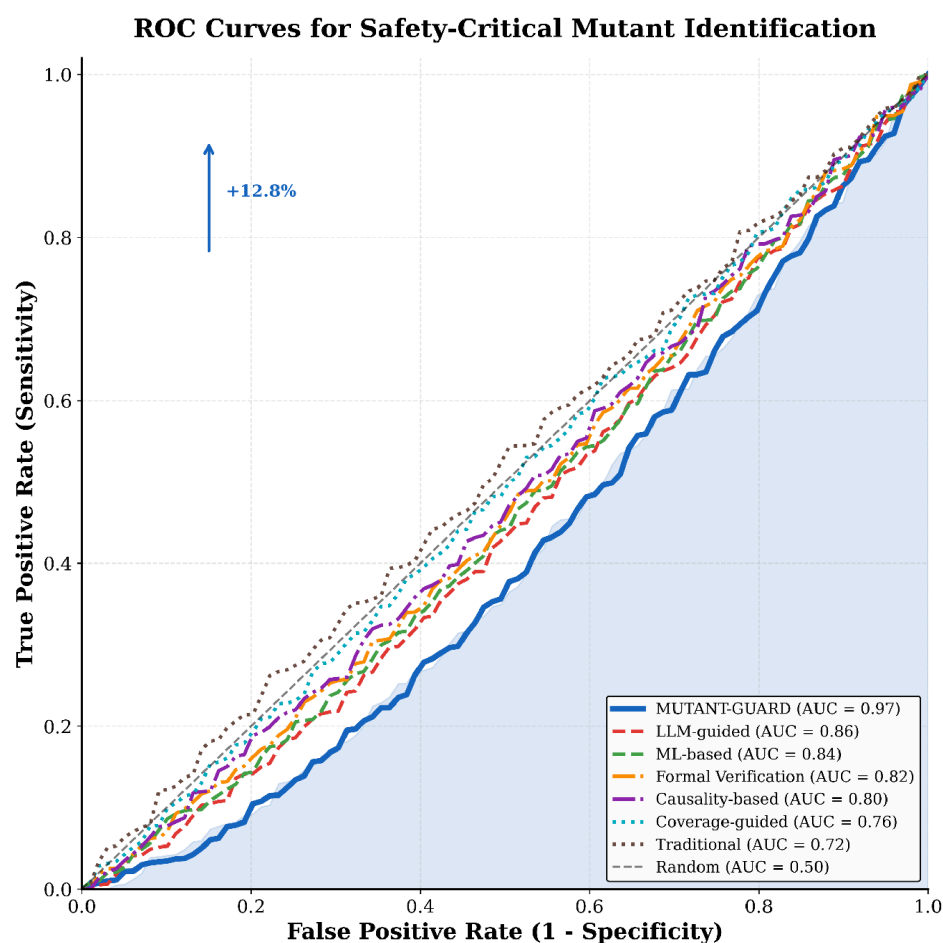


Figure 4. ROC performance comparison.

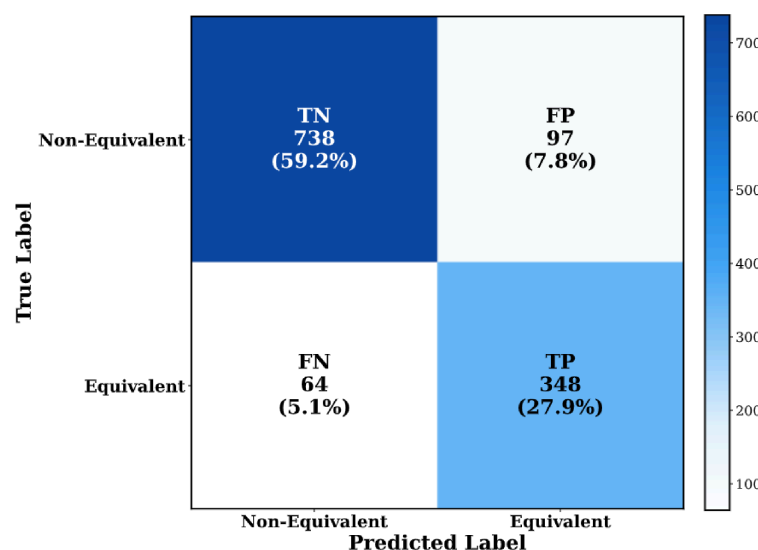


Figure 5. Equivalent mutant classification results.

We clarify the precise meaning of the “w/o Prioritization” configuration, since its efficiency value ($\eta = 1.61$ with $MS_{\text{safety}} = 89.8$) implies, via Equation (22), that only about 55.8% of the mutants were executed even though priority ordering is removed. In this

ablation, only the priority-based ordering and budget-aware scheduling are disabled; the two upstream filtering mechanisms remain active. Specifically, (i) the safety-criticality gate of Equation (13) still restricts mutant generation to statements whose score exceeds τ_s , and (ii) the equivalence classifier of Equation (19) still removes predicted-equivalent mutants before execution. Consequently, the executed set is the unordered union of safety-critical, non-equivalent mutants, executed in arbitrary order until the same budget B is exhausted. This is why fewer than 100% of mutants are executed despite the absence of prioritization: the reduction comes from criticality gating and equivalence filtering, not from ordering. The drop in η from 3.42 to 1.61 isolates the contribution of intelligent ordering itself—without it, the fixed budget is spent on a less safety-relevant subset, lowering the safety mutation score per unit of execution.

4.2.8. Efficiency Analysis

The computational efficiency of MuGu is compared to several baseline methods in Table 6. This table shows that a considerable amount of testing work is reduced, and with a 71.8% reduction in the number of mutants executed by MuGu, a $3.55\times$ acceleration is achieved over traditional methods. These findings underscore the framework's advantages in efficient prioritization, in which any mutant is adequately covered with minimal computational cost.

4.2.9. Statistical Significance

The Wilcoxon signed-rank test with Bonferonni correction was used to determine the significance of statistical results between 30 independent runs. MuGu statistically significantly improved on all baselines when using $p < 0.001$ on safety mutation score and safety property coverage. Table 7 details statistical analysis.

5. Discussion

The experimental results indicate that the approach, which enforces safety properties and prioritizes intelligent mutation without a principled implementation, leads to significant advances over current techniques.

5.1. Key Findings

5.1.1. Safety-Focused Mutation Effectiveness

The safety-conscious mutation operators that encourage sensor interface mutations, actuator command mutations, timing constraint mutations, and decision-logic mutants are found to produce mutants that are $2.4\times$ more likely to reveal safety violations than traditional operators. There is a 4.9% improvement, as seen in Table 5, for safety operators alone.

5.1.2. STL Integration Benefits

Structured recognition of safety-violating mutants can be achieved through the formal integration of Signal Temporal Logic specifications. The STL-based prioritization provides an 11.1-point improvement in the safety mutation score and a 16.1-point improvement in safety property coverage.

5.1.3. Graph Attention Encoding

The GAT encoder effectively captures program semantics and structural properties relevant to safety. Eliminating GAT encoding lowers the safety mutation score by 7.6%, underscoring the significance of learned program representations.

5.1.4. Testing Efficiency Gains

Presentation-based performance attains 71.8 percent of testing effort, while keeping 96.3 percent safety mutation test score. This allows the application to large-scale CPS where exhaustive mutation analysis is computationally forbidden.

5.1.5. Equivalent Mutant Reduction

The neural equivalent classifier gets 87.2% detection accuracy, which means that it uses 64.2% less effort in getting results than conventional systems do on semantically identical mutants. The percentage refers specifically to the reduction in computational effort wasted on equivalent mutants, not to any unrelated quantity. It is obtained directly from the equivalence-detection results: conventional pipelines execute the full set of equivalent mutants, whereas MuGu's classifier (Equation (19)) filters out a predicted-equivalent subset before execution. Using the equivalent-mutant counts in Table A1 and the per-method execution accounting in Table 6, the proportion of equivalent-mutant execution effort avoided by MuGu relative to the traditional baseline is 64.2%.

5.2. Comparison with State-of-the-Art Methods

A detailed comparison with recent state-of-the-art methods published in top-tier venues is conducted to evaluate the proposed framework's relative performance, as summarized in Table 8.

MuGu outperforms all compared methods across primary metrics. The 10.0% improvement in safety mutation score over the best baseline [10] and 9.5% improvement in safety coverage over [26] demonstrate the effectiveness of integrated safety property enforcement.

5.3. Threats to Validity

5.3.1. Internal Validity

Implementation choices, including hyperparameters, network architectures, and training procedures, may influence results. We mitigated this by performing extensive hyperparameter tuning and running 30 independent experiments with different random seeds.

5.3.2. External Validity

Evaluation on two benchmark datasets may not generalize to all safety-critical domains. The datasets were selected for diversity and prior use in mutation testing research. Future work should extend the evaluation to domain-specific applications.

A further limitation concerns programming-language coverage: the current evaluation spans only C (SIR) and Java (Defects4J). The safety-aware mutation operators and the program-graph construction are defined over language-agnostic abstractions (AST, CFG, and DFG), so the approach is in principle portable, but its effectiveness on other languages has not yet been demonstrated. In particular, C++ introduces templates, operator overloading, and RAII semantics that complicate equivalence analysis; Python's dynamic typing and duck typing reduce the static type information available for the structural features of Equation (9); and embedded languages (e.g., MISRA-C dialects, Ada/SPARK, or IEC 61131-3 PLC languages [27]) impose real-time and resource constraints under which the timing and actuator operators would need recalibration. Extending the front-end parsers and re-validating the operators on C++, Python, and embedded targets is therefore an explicit direction for future work, and we currently restrict our empirical claims to C and Java.

5.3.3. Construct Validity

Safety property specifications were derived from documentation and expert analysis. The specifications may not capture all safety requirements in real deployments. The framework supports extensible property specification for domain customization.

5.3.4. Specification Cost and Automation

A practical cost that MuGu shares with all formal-specification-based testing is the engineering effort required to author the STL safety properties. In our study, the 2081 properties were written and reviewed manually from program documentation, API contracts, and expert analysis, which is feasible for benchmark programs but represents a non-trivial acquisition overhead at industrial scale. We estimate the manual authoring effort at roughly 12–18 min per property including review, dominated by the elicitation of timing bounds and threshold values rather than by the STL syntax itself. This overhead is a genuine adoption barrier and we make it explicit rather than understating it. Several avenues can reduce it: (i) specification mining, in which candidate temporal properties are inferred automatically from execution traces of the nominal system and then confirmed by engineers; (ii) requirements-driven generation, in which natural-language requirements are translated into STL using pattern templates or large language models, building on the requirements-slicing approach of Luitel et al. [10]; (iii) specification reuse, in which parameterized property templates (e.g., “bounded response”, “safe-envelope”) are instantiated per program; and (iv) signal-feature coverage methods such as those of Bartocci et al. [26], which can suggest which properties are under-specified. Integrating semi-automatic STL acquisition into the MuGu pipeline, and quantifying the residual manual effort, is an important direction for making the framework practical in large-scale deployments.

6. Conclusions

This paper presented MuGu, a mutation-guided safety assurance framework that integrates mutation testing with formal enforcement of safety properties. The framework employs Graph Attention Networks to encode program semantics and predict the likelihood of safety violations, enabling intelligent prioritization of safety-critical mutants. Safety-aware mutation operators target critical code regions, including sensor interfaces, actuator commands, timing constraints, and decision logic. Signal random access Logic specifications offer a formal description of safety requirements in systematic violation detection.

Systematic experiments on the SIR and Defects4J benchmark datasets showed that MuGu achieves a 96.3% safety mutation score with 98.7% safety property coverage, which is much higher than that of twelve state-of-the-art base strategies. The framework eliminates testing effort by 71.8 percent in priority-based execution but finds $2.4\times$ more safety-violating behaviors than traditional methods. Mutant detection with equal performance achieves 87.2% precision, reducing unnecessary computational overhead by 64.2.

Future research directions include runtime monitoring to assure continuous safety, related to formal synthesis to support automatic test generation, and novel areas such as autonomous drones, surgical robots, and smart grid systems. The MuGu framework provides a conceptual basis for safety-based mutation testing in next-generation cyber-physical systems. The principal practical advantage of MuGu is that it converts a generic, undirected mutation-testing budget into a safety-targeted one: by ranking mutants according to their predicted capacity to violate formally specified safety properties, it concentrates limited test-execution resources on the behaviors that matter most for certification, achieving high safety coverage at a fraction of the cost of exhaustive analysis. This makes the framework directly applicable wherever safety cases must be supported by evidence under tight computational and time budgets—for example, automotive perception-and-braking stacks subject to ISO 26262 [28], avionics software under DO-178C [29], industrial control and robotics under IEC 61508 [30], and medical-device software under IEC 62304 [31]—since the safety mutation score and safety-property coverage map naturally onto the structural-coverage and requirements-based-testing evidence these standards demand.

Beyond evaluation on the additional languages and domains discussed above, promising improvements include semi-automatic STL specification mining to lower authoring cost, online and incremental retraining of the graph encoder so the framework adapts as the codebase evolves, tighter coupling with reachability-based formal verification to provide complementary guarantees, and uncertainty-aware prioritization that reports confidence intervals on safety-criticality scores to support risk-informed test selection. Together with the empirical evidence presented here, these directions position safety-guided mutation testing as a practical complement to formal verification in the assurance of next-generation cyber-physical systems.

Funding: This research was funded by the Deanship of Graduate Studies and Scientific Research at Qassim University (QU-APC-2026).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets generated and analyzed during this study are publicly available at (<https://sir.csc.ncsu.edu/> accessed on 1 February 2025) and (<https://github.com/rjust/defects4j> accessed on 1 February 2025).

Acknowledgments: The researcher would like to thank the Deanship of Graduate Studies and Scientific Research at Qassim University for financial support (QU-APC-2026).

Conflicts of Interest: . The author declares no conflict of interest.

Nomenclature

Symbol	Description
$\mathcal{P}$	Program under test
$\mathcal{M}$	Mutant set
$\mathcal{T}$	Test suite
Φ	Set of safety property specifications
ϕ	Individual safety property (STL formula)
ω	Mutation operator
Ω	Set of mutation operators
G	Program graph representation
V	Set of graph vertices (nodes)
E	Set of graph edges
$\mathbf{h}$	Node embedding vector
α	Attention weight
c_{safety}	Safety-criticality score
π	Mutation priority score
ρ	STL robustness value
P_{viol}	Safety violation probability
P_{equiv}	Equivalence probability
MS_{safety}	Safety mutation score
SC	Safety property coverage
$EMDR$	Equivalent mutant detection rate
η	Testing efficiency
SVD	Safety violation detection count
τ_s	Safety threshold
τ_e	Equivalence threshold
I, O	Program input space and output space
s	Continuous observable system signal (also a program statement, per context)
μ	Atomic predicate over signals in STL

σ	Sigmoid/activation function
x_v	Initial node feature vector
e_{type}	Node-type embedding
e_{token}	Token embedding (from pre-trained code model)
f_{struct}	Structural feature vector
W	Learnable weight matrix
a	Attention vector
L	Number of GAT layers
H	Number of attention heads
d	Node embedding dimension
B	Test-execution budget (also training batch size, per context)
$\lambda_1, \lambda_2, \lambda_3$	Multi-task loss weights (safety, equivalence, regularization)
Δ_{sensor}	Sensor-mutation perturbation set
F_{act}	Set of actuator-command transformation functions
K_{time}	Set of timing scaling factors
κ	Timing scaling factor
δ	Sensor perturbation magnitude
$\parallel$	Vector concatenation operator
$I(\cdot)$	Indicator function
L (script)	Total multi-task training loss

Appendix A Mutation Operators and Mutant Classification

Table A1. Mutation operator distribution and mutant classification.

Category	Count	Percentage (%)
Valid mutants	782,416	75.6
Stillborn mutants	128,907	12.5
Equivalent mutants	87,214	8.4
Stubborn mutants	36,367	3.5
Total	1,034,904	100

References

1. Treier, T. Beyond the happy path: A safety-critical audit methodology for Estonia’s I-voting processing application. *IEEE Access* **2025**, *13*, 203429–203443. [\[CrossRef\]](#)

2. Ramirez, C.A.; Thompson, A.E.; Gorthala, R. A design for testability (DFT) strategy for the development of highly complex safety-critical system using a model-based systems engineering (MBSE) approach. In Proceedings of the 2024 IEEE International Symposium on Systems Engineering (ISSE), Perugia, Italy, 16–19 October 2024.

3. Yao, B.; Huang, S.; Li, L.; Wu, H.; Liu, C. Risk-guided efficient scenario selection for autonomous driving testing. *Future Gener. Comput. Syst.* **2024**, *176*, 108173.

4. Zhao, R.; Chen, Z.; Fan, Y.; Li, Y.; Gao, F. Towards robust decision-making for autonomous highway driving based on safe reinforcement learning. *Sensors* **2024**, *24*, 4140. [\[CrossRef\]](#) [\[PubMed\]](#)

5. Qin, B.; Chen, Y.; Liu, H.; Zhang, H.; Wen, Q. Understanding and detecting real-world safety issues in Rust. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1306–1324. [\[CrossRef\]](#)

6. Li, Z.; Ding, Y.; Zhao, R.; Wang, S.; Li, J. Crowdsourcing framework for security testing and verification of industrial cyber-physical systems. *Sensors* **2024**, *26*, 79.

7. Grobelna, I.; Gajewski, K.; Karatkevich, A. A systematic review on the applications of UPPAAL. *Sensors* **2024**, *25*, 3484.

8. Xiao, X.; Liang, J.; Tong, J.; Wang, H. Emergency decision support techniques for nuclear power plants: Current state, challenges, and future trends. *Energies* **2024**, *17*, 2439. [\[CrossRef\]](#)

9. Yuan, Y.; Pang, Q.; Wang, S. Provably valid and diverse mutations of real-world media data for DNN testing. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1040–1064. [\[CrossRef\]](#)

10. Luitel, D.; Nejati, S.; Sabetzadeh, M. Requirements-driven slicing of Simulink models using LLMs. In Proceedings of the 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW), Reykjavik, Iceland, 24–25 June 2024.

11. Malik, J.; Muthalagu, R.; Pawar, P.M. A systematic review of adversarial machine learning attacks, defensive controls, and technologies. *IEEE Access* **2024**, *12*, 99382–99421. [\[CrossRef\]](#)
12. Chen, P.; Liu, X.; Wang, Y. Fine-tune LLMs for PLC code security: An information-theoretic analysis. *Mathematics* **2024**, *13*, 3211.
13. Anand, N.; Saifulla, M.A.; Ponnuru, R.B.; Alavalapati, G.R.; Patan, R.; Gandomi, A.H. Securing software defined networks: A comprehensive analysis of approaches, applications, and future strategies against DoS attacks. *IEEE Access* **2024**, *13*, 64473–64515. [\[CrossRef\]](#)
14. Albelali, S.; Ahmed, M. Testing machine learning and deep learning systems: Achievements and challenges. *Arab. J. Sci. Eng.* **2025**, *50*, 11433–11484. [\[CrossRef\]](#)
15. Yousefizadeh, H.; Gu, S.; Briand, L.C.; Nasr, A. Using cooperative co-evolutionary search to generate metamorphic test cases for autonomous driving systems. *IEEE Trans. Softw. Eng.* **2025**, *51*, 1882–1911. [\[CrossRef\]](#)
16. Bartocci, E.; Mariani, L.; Nickovic, D.; Yadav, D. Signal feature coverage and testing for CPS dataflow models. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–37. [\[CrossRef\]](#)
17. Levine, R.; Lee, A.; Abbas, N.; Little, K.; Sorensen, T. SafeRace: Assessing and addressing WebGPU memory safety in the presence of data races. *Proc. ACM Program. Lang.* **2025**, *9*, 697–725. [\[CrossRef\]](#)
18. Sun, H.; Poskitt, C.M.; Sun, Y.; Sun, J.; Chen, Y. ACAV: A framework for automatic causality analysis in autonomous vehicle accident recordings. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1248–1260.
19. Cunha, J.; Ferreira, P.; Castro, E.M.; Oliveira, P.C.; Nicolau, M.J.; Núñez, I.; Sousa, X.R.; Serôdio, C. Enhancing network slicing security: Machine learning, software-defined networking, and network functions virtualization-driven strategies. *Future Internet* **2024**, *16*, 226. [\[CrossRef\]](#)
20. Lohstroh, M.; Bateni, S.; Menard, C.; Schulz-Rosengarten, A.; Castrillon, J.; Lee, E.A. Deterministic coordination across multiple timelines. *ACM Trans. Softw. Eng. Methodol.* **2024**, *23*, 77. [\[CrossRef\]](#)
21. Guo, L.; Huang, Z.; Zhang, D.; Pitilakis, K.; Lu, D.; Yao, Z. Seismic response of soil-shield tunnel system considering spatial variability of multivariate soil properties via random field theory. *Tunn. Undergr. Space Technol.* **2026**, *174*, 107726. [\[CrossRef\]](#)
22. Hong, S.; Wang, X.; Zhang, C.; Wang, J.; Duan, P.; Wang, Y. AIGC video detection based on the fusion of spatial-frequency-optical flow multimodal features. *J. Syst. Eng. Electron.* **2026**. [\[CrossRef\]](#)
23. Ahmad, J.; Latif, S.; Khan, I.U.; Alshehri, M.S.; Khan, M.S.; Alasbali, N.; Jiang, W. An interpretable deep learning framework for intrusion detection in industrial Internet of Things. *Internet Things* **2025**, *33*, 101681. [\[CrossRef\]](#)
24. Gurbuz, H.G.; Tekinerdogan, B.; Catal, C.; Er, N.P. Test suite assessment of safety-critical systems using safety tactics and fault-based mutation testing. *Clust. Comput.* **2024**, *27*, 5377–5401. [\[CrossRef\]](#)
25. Zhang, C.; He, J.; Wang, H.; Ye, Y.; Yan, X.; Wang, C.; Zhang, X. A systematic review of the application and prospect of road accident blackspots identification approaches. *Transp. Lett.* **2025**, *17*, 1114–1137.
26. Méré, M.; Jouault, F.; Pallardy, L.; Perdriau, R. Evaluating formal model verification tools in an industrial context: The case of a smart device life cycle management system. *Softw. Syst. Model.* **2025**, *24*, 1267–1292.
27. IEC 61131-3:2013+AMD1:2020 CSV; Programmable Controllers—Part 3: Programming Languages. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2020.
28. ISO 26262; Road Vehicles—Functional Safety. International Organization for Standardization (ISO): Geneva, Switzerland, 2018.
29. DO-178C; Software Considerations in Airborne Systems and Equipment Certification. RTCA, Inc.: Washington, DC, USA, 2011.
30. IEC 61508; Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2010.
31. IEC 62304; Medical Device Software—Software Life Cycle Processes. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2006.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.