Gazi University
Journal of Science
PART A: ENGINEERING AND INNOVATION
http://dergipark.org.tr/gujsa

Graph Neural Network-Based Prediction of Soft Error Vulnerability and Criticality of Functions in Scientific Applications

Sanem ARSLAN[1*]

[1] *Marmara University, Faculty of Engineering, Department of Computer Engineering, İstanbul, Türkiye*

Keywords
Fault Tolerance
Reliability
GNN
Vulnerability Prediction

Abstract

Soft errors caused by transient hardware faults can lead to silent data corruptions (SDCs) in scientific applications, potentially impacting correctness and reliability. Traditional fault injection (FI) methods provide accurate vulnerability measurements but are prohibitively time-consuming and resource-intensive. In this work, we propose a function-level prediction framework for SDC vulnerability and criticality in CPU-based scientific applications using Graph Neural Networks (GNNs). Static code features are extracted from LLVM intermediate representation and used to construct function call graphs, enabling GCN, GAT, and GraphSAGE models to capture both intra-function characteristics and inter-function dependencies. The problem is formulated as both regression and classification, predicting continuous vulnerability and criticality scores as well as binary labels. The evaluation is conducted on 30 applications (90 functions) from the PolyBench benchmark suite using leave-one-application-out cross-validation, ensuring that the model is tested on unseen applications. Among the evaluated architectures, GraphSAGE achieves the highest performance (F1 = 0.80, MAE = 0.17), showing strong generalization across diverse workloads. Feature correlation and model-based importance analyses identify the most influential LLVM features, and results demonstrate that the proposed approach provides fine-grained, accurate predictions without the need for exhaustive FI campaigns, enabling more efficient and targeted fault-tolerance strategies.

Cite

Arslan, S. (2025). Graph Neural Network-Based Prediction of Soft Error Vulnerability and Criticality of Functions in Scientific Applications. *GU J Sci, Part A*, *12*(4), 979-998. doi:10.54287/gujsa.1766028

Author ID (ORCID Number)		Article Process	
0000-0003-3019-7070	Sanem ARSLAN	**Submission Date**	19.08.2025
		Revision Date	07.10.2025
		Accepted Date	14.11.2025
		Published Date	31.12.2025

1. INTRODUCTION

As computing systems become increasingly complex and pervasive, ensuring their reliability in the presence of transient hardware faults has become a critical challenge. Such faults, often caused by environmental factors like cosmic rays or voltage fluctuations, can lead to soft errors, which may propagate silently through computations and result in incorrect program outputs. In scientific and safety-critical applications, these silent data corruptions (SDCs) can have severe consequences. SDC vulnerability is commonly measured through fault injection (FI) campaigns (Lu et al., 2015), statistical fault analysis, or architectural vulnerability factor (AVF) estimation (Mukherjee et al., 2002). While FI provides high-fidelity results, it requires executing numerous instrumented test runs and is therefore extremely time-consuming and resource-intensive, making predictive approaches an attractive alternative. In typical resilience studies, a single FI campaign may require

*Corresponding Author, e-mail: sanem.arslan@marmara.edu.tr

tens of hours per benchmark to complete thousands of fault injections, motivating the development of predictive methods that can provide comparable insight with significantly lower computational cost.

Accurate vulnerability prediction allows designers to identify critical code regions and apply targeted protection mechanisms, thereby reducing performance and energy overhead compared to full protection strategies. Previous work has explored machine learning approaches for this task, typically at the application or kernel level (Öz & Arslan, 2021; Topçu & Öz, 2023; Wei et al., 2023). While effective in reducing the need for costly FI campaigns, these coarse-grained approaches cannot capture the fine-grained variations in vulnerability that occur between functions within the same application. Region- or function-level prediction can provide more targeted and efficient fault tolerance, as protection mechanisms can be selectively applied to the most vulnerable parts of the code rather than the entire application (Arslan & Unsal, 2021). Previous studies have demonstrated that selective protection of only the most critical code regions can achieve substantial resilience improvements with limited performance and energy overhead. For instance, selectively replicating the most error-sensitive functions can reduce the silent data corruption (SDC) rate by up to 7.6×, while incurring only about 22% performance and 37% energy overhead compared with full redundant execution (Arslan & Unsal, 2021). In recent years, modern deep learning methods, particularly Graph Neural Networks (GNNs), have been successfully applied to program analysis and vulnerability detection (Allamanis et al., 2018; Zou et al., 2021), offering promising capabilities to model both structural and semantic relationships in code.

In this work, we propose a function-level soft error vulnerability and criticality prediction framework that leverages static program analysis and modern deep learning techniques. Using the LLVM compiler infrastructure, we extract a rich set of static features from each function, including structural, control-flow, and data-flow characteristics. We then construct graph representations of functions and employ Graph Neural Networks (GCNs, GATs, GraphSAGE) to model dependencies within the code, enabling a more detailed and context-aware vulnerability analysis. This approach eliminates the need for exhaustive FI while providing finer-grained predictions that can inform selective protection strategies.

The key contributions of this paper are as follows:

- We propose a function-level soft error vulnerability and criticality prediction framework for CPU-based scientific applications, combining LLVM-based static feature extraction with Graph Neural Network modeling of function call graphs.
- We formulate the problem as both regression and classification, enabling prediction of continuous scores and binary labels, and evaluate the approach on 30 PolyBench applications using leave-one-out cross-validation, achieving high accuracy and low prediction error.
- We analyze feature importance via correlation-based selection, identifying the most influential LLVM features and assessing model robustness to reduced feature sets.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides background information on soft error vulnerability and the metrics targeted for prediction in this study. Section 4 describes our proposed methodology, including feature extraction and GNN model design. Section 5 presents the experimental setup and evaluation results. Finally, Section 6 concludes the paper and discusses future research directions.

2. RELATED WORK

The use of machine learning techniques to predict soft error vulnerability has been studied in the literature.

Öz and Arslan (2021) utilized classical machine learning techniques to estimate the soft error vulnerability of parallel applications, leveraging application features to predict error susceptibility without requiring exhaustive fault injection, which significantly reduced evaluation cost. Unlike their application-level vulnerability prediction approach, our work targets function-level prediction, enabling finer-grained analysis through modern deep learning techniques such as Graph Neural Networks.

Topçu and Öz (2023) proposed a machine learning–based approach for predicting the soft error vulnerability of GPGPU applications. Their method extracts a set of application-level features from GPU kernels and trains classical machine learning models to estimate error susceptibility, aiming to reduce the need for costly and time-consuming fault injection experiments. While their study focuses on coarse-grained, application-level predictions in the context of massively parallel GPU workloads, our work targets fine-grained, function-level vulnerability prediction for CPU applications. Moreover, we employ modern deep learning methods, specifically Graph Neural Networks, to model structural and data-flow dependencies within application functions, enabling a detailed and context-aware analysis.

Wei et al. (2023) introduced a graph learning–assisted model (GLAM-SERP) designed to predict the soft error resilience of GPGPU applications. Their approach constructs graph representations of GPU kernels and employs graph neural networks to capture structural dependencies, enabling accurate prediction of resilience without exhaustive fault injection. GLAM-SERP focuses on GPU-specific workloads and models resilience at the instruction level, with an emphasis on massively parallel architectures. In contrast, our work targets function-level vulnerability prediction for general CPU applications. While both studies employ graph-based deep learning, our methodology integrates program structure and feature extraction within the LLVM framework to achieve finer-grained analysis.

Ni et al. (2024) proposed a function-level vulnerability detection framework that fuses multi-modal knowledge, combining code structure, semantics, and runtime information to improve detection accuracy. While their approach is focused on security vulnerabilities in general-purpose software, our method addresses resilience against transient hardware faults in scientific applications, using only static features extracted via LLVM.

Laguna et al. (2016) proposed a dynamic protection framework for scientific applications that selectively applies error detection mechanisms to code regions most susceptible to silent data corruptions (SDCs). Their approach uses profiling and lightweight analysis to identify vulnerable parts of the code and applies protection only where it is most beneficial. While the study focuses on runtime detection and mitigation of SDCs at the instruction level, our work addresses the problem from a predictive modeling perspective, aiming to estimate function-level vulnerability in advance.

Cao et al. (2022) presented a memory-related vulnerability detection framework that leverages flow-sensitive Graph Neural Networks to identify memory errors such as buffer overflows and use-after-free bugs. Their approach combines control-flow and data-flow graph analysis to capture contextual information. Although our work also applies GNNs for vulnerability prediction, we focus on resilience against transient hardware faults rather than software security flaws.

3. BACKGROUND

3.1. Soft Error Vulnerability

Soft errors are transient faults in digital circuits caused by radiation-induced particle strikes, voltage fluctuations, or other environmental factors that do not cause permanent hardware damage but can alter the system state. These faults can lead to Silent Data Corruptions (SDCs) if they change computation results without causing a system crash, or they can cause detectable failures if they violate program control flow or memory access rules. Soft error vulnerability quantifies the likelihood that a fault in a given hardware or software component will result in incorrect output. In software-level analyses, vulnerability is often expressed in terms of the proportion of faults in a program region that propagate to the final output. Understanding and mitigating this vulnerability is particularly critical for high-performance computing and safety-critical systems, where even small error rates can have significant consequences.

3.2. Vulnerability and Criticality Results of Applications

To better understand the error characteristics of target workloads, fault injection experiments are performed on 30 compute-intensive scientific applications from the PolyBench benchmark suite (Pouchet, 2012) in a previous study (Arslan & Unsal, 2021). In the study, the vulnerability of each application function is evaluated based on a high number of fault injection experiments. Each application consists of a small set of functions, init_array, main_algorithm, and print_array, which handle initialization, core computation, and output, respectively. The main parameters of the FI setup are summarized here. Faults were injected using the LLFI (LLVM-Level Fault Injector) framework, which operates at LLVM's intermediate representation (IR) level. A single-bit transient fault model was adopted, where one randomly selected instruction output (destination register) was flipped per injection. Each application from the PolyBench benchmark suite underwent 10,000 independent injections, distributed proportionally to the execution-time share of each function. Injection sites and timing were randomized per trial using unique random seeds. After each run, outcomes were classified as Masked, Silent Data Corruption (SDC), or Crash, based on bit-wise comparison with a golden output from error-free execution. Each application consists of a small set of functions, init_array, main_algorithm, and print_array, which handle initialization, core computation, and output, respectively. For each function, two key metrics are evaluated:

- SDC Vulnerability: The fraction of injected faults in a given function that result in silent data corruptions (SDCs) at the application output. SDC occurs when a hardware or software fault causes incorrect application output without any failure notification.
- Criticality: Computed as the product of the function's SDC rate (normalized between 0 and 1) and its execution time percentage within the application (also normalized between 0 and 1).

While SDC vulnerability reflects only the inherent error-proneness of a function, criticality incorporates both fault susceptibility and performance impact, providing a more comprehensive measure of the function's importance in the overall application. The detailed SDC vulnerability and criticality results for each function in the PolyBench suite are presented in Table 1 and 2, respectively.

The results show that, for most applications (23 out of 30), the init_array function exhibits the highest SDC rate. Since this function initializes the application's input, a fault occurring here can easily propagate to the final output. Fault probability is also strongly correlated with execution time, as functions with longer execution durations have a higher likelihood of experiencing a fault. Therefore, criticality becomes an important metric to capture the combined effect of fault susceptibility and performance impact. Although the main_algorithm function does not have the highest SDC rate in most cases, it is identified as the most critical function in 25 out of 30 applications. For further details of fault injection experiments and SDC vulnerability and criticality analysis, we redirect the reader to the previous study (Arslan & Unsal, 2021).

As fault injection experiments are time-consuming and computationally expensive, the ability to predict SDC vulnerability and criticality using static analysis and machine learning is highly advantageous. The patterns observed in this analysis motivate the methodology described in Section 4, where we leverage static code features and graph-based learning to model and predict function-level vulnerability and criticality.

Table 1. *Relative SDC rates of functions in Polybench applications (Arslan & Unsal, 2021)*

Application Name	**Function Name**		
	init_array	**main algorithm**	**print array**
bicg	56.28%	29.16%	40.00%
2mm	49.19%	29.77%	40.00%
3mm	51.31%	25.30%	37.04%
atax	57.80%	34.52%	44.00%
doitgen	48.68%	27.77%	25.66%
mvt	55.47%	34.48%	36.17%
trmm	64.73%	33.55%	35.06%
gemm	50.75%	27.11%	29.85%
gemver	52.75%	34.00%	46.67%
gesummv	56.26%	32.37%	41.82%
symm	58.23%	33.96%	34.79%
syr2k	62.50%	32.82%	35.03%
syrk	55.39%	32.32%	32.78%
cholesky	12.82%	42.44%	40.79%
durbin	59.32%	41.63%	43.41%
ludcmp	13.61%	29.36%	50.00%
lu	21.84%	41.09%	31.01%
trisolv	57.26%	35.83%	43.33%
correlation	45.56%	25.15%	28.41%
covariance	50.47%	31.11%	30.83%
deriche	46.47%	31.22%	32.40%
nussinov	0.45%	14.09%	47.59%
adi	61.70%	16.85%	26.53%
fdtd-2d	46.88%	31.89%	26.56%
heat-3d	39.34%	10.60%	25.24%
jacobi-1d	27.44%	19.16%	36.09%
jacobi-2d	33.52%	24.74%	32.00%
seidel-2d	67.26%	34.94%	38.60%
floyd_warshall	26.13%	5.75%	36.00%
gramschmidt	28.92%	36.22%	30.61%

Table 2. *Criticality results of PolyBench application functions (Arslan & Unsal, 2021)*

Application Name	**Function Name**		
	init_array	**main algorithm**	**print array**
bicg	0.167	0.202	0.004
2mm	0.036	0.270	0.007
3mm	0.027	0.237	0.004
atax	0.160	0.248	0.002
doitgen	0.031	0.241	0.018
mvt	0.152	0.247	0.003
trmm	0.052	0.288	0.021
gemm	0.037	0.246	0.006
gemver	0.092	0.279	0.001
gesummv	0.236	0.185	0.002
symm	0.046	0.300	0.013
syr2k	0.048	0.291	0.012
syrk	0.050	0.277	0.018
cholesky	0.109	0.061	0.003
durbin	0.007	0.405	0.006
ludemp	0.107	0.063	0.000
lu	0.163	0.098	0.004
trisolv	0.272	0.182	0.006
correlation	0.114	0.121	0.082
covariance	0.152	0.162	0.088
deriche	0.032	0.266	0.025
nussinov	0.000	0.135	0.009
adi	0.003	0.167	0.001
fdtd-2d	0.009	0.305	0.006
heat-3d	0.005	0.104	0.003
jacobi-1d	0.006	0.184	0.005
jacobi-2d	0.006	0.241	0.003
seidel-2d	0.008	0.342	0.004
floyd_warshall	0.003	0.056	0.003
gramschmidt	0.012	0.330	0.015

4. METHODOLOGY

The methodology is designed to enable accurate prediction of both soft error vulnerability and function criticality in scientific applications by jointly leveraging static code features and inter-function relationships. An overview of the proposed framework is illustrated in Figure 1. In this study, we used 30 computationally intensive scientific applications from the PolyBench suite. For every function in the PolyBench applications, we extract a total of 12 static code features using custom LLVM passes. These features include counts of function calls, stack allocations, global variable accesses, floating-point operations, and several other code-level metrics. These features are described in Section 4.1. Next, we construct a graph representation of each application, where nodes correspond to functions, node attributes are the extracted feature vectors, and edges are derived from the function call graph. We then train three GNN models, namely Graph Convolutional Network (GCN), Graph Attention Network (GAT), and GraphSAGE for both regression and classification tasks, as described in Section 4.2. The models are trained with leave-one-out cross-validation applied across the applications. Model performance is evaluated using mean absolute error (MAE) and root mean squared error (RMSE) for regression, and accuracy, precision, recall, and F1 score for classification. Furthermore, we conduct feature selection experiments to examine the influence of the most correlated features on predictive performance.

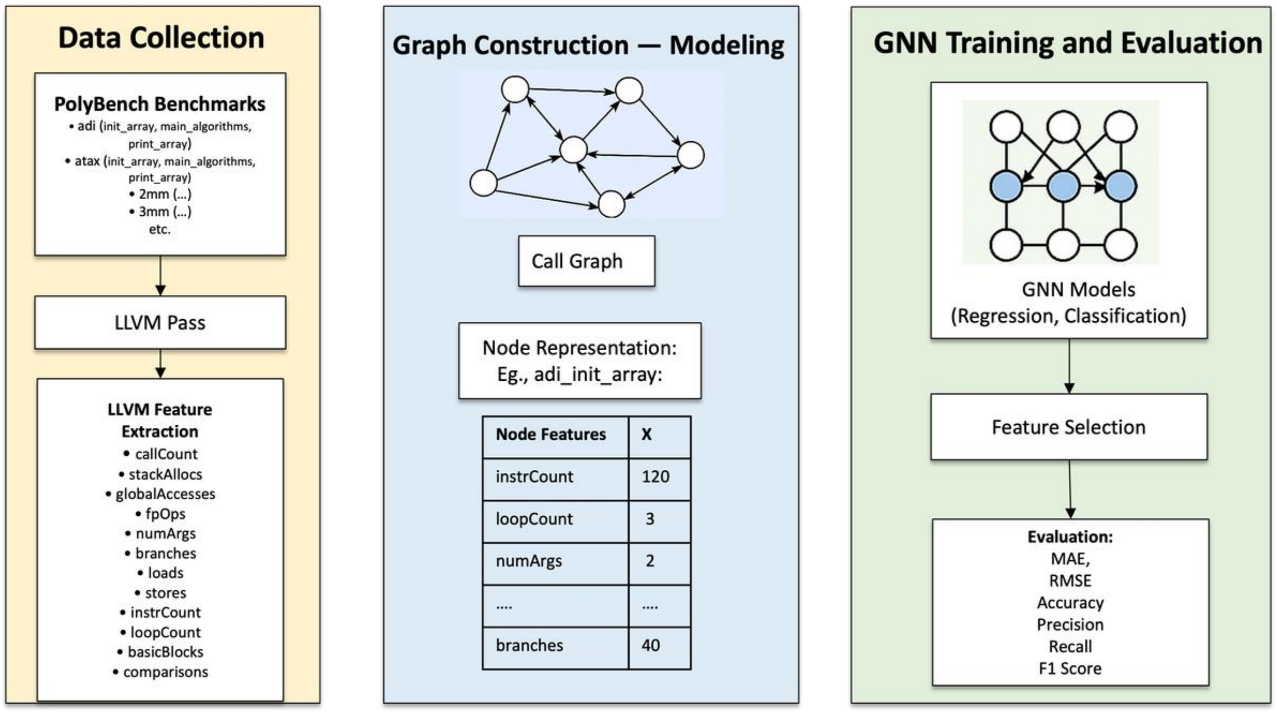

Figure 1. *General overview of our framework*

The details of each step are explained in the following subsections.

4.1. Data Collection

A total of 12 distinct static features were extracted using the custom LLVM FunctionPass, as listed in Table 3. These LLVM-level features are closely related to the fault tolerance characteristics of scientific applications, as they capture both computational and memory access behaviors that influence error propagation. Features such as *fpOps*, *loads*, and *stores* directly affect the likelihood and impact of soft errors, since bit flips in floating-point computations or memory operations can cause significant numerical deviations or persistent data corruption. Control-flow related features, including *branches* and *loopCount*, influence how faults manifest, with erroneous branch decisions potentially redirecting execution and loops increasing the effect of a single error over multiple iterations. Memory allocation patterns, captured by *stackAllocs* and *globalAccesses*, determine the scope and lifetime of vulnerable data, with global variables being particularly critical as they are accessible across multiple functions. Finally, *callCount*, *numArgs*, and *instrCount* reflect functional complexity and dependency chains, which can increase the propagation paths of errors. By quantifying these aspects, the extracted features provide meaningful indicators for predicting both soft error vulnerability and the criticality of functions.

To collect these features, we implemented a custom LLVM FunctionPass in C++ that analyzes the LLVM Intermediate Representation (IR) of each function in the PolyBench benchmarks. The pass extracts static code characteristics, including counts of instructions, memory operations, and control flow properties, and also records function call relationships to construct the call graph. We chose LLVM IR for feature extraction because it provides a rich, language-independent, and compiler-optimized view of the program, enabling precise and consistent analysis across all benchmarks.

The feature extraction was implemented using the LLVM 17 toolchain (Apple Clang 17.0.0) on macOS. The pass traverses each function’s IR and counts instruction categories, control-flow structure, and memory-access operations. Loop information is collected via LLVM’s LoopAnalysis, enumerating loops recursively so that nested loops are included; hence, *loopCount* represents the total number of loops, not merely the maximum nesting depth.

While the current implementation represents the call graph as unweighted, directed edges between caller and callee functions, future extensions will incorporate inter-procedural and graph-topological descriptors to enrich this representation. Although such metrics (e.g., edge weights, in-degree, out-degree, or PageRank centrality) may have limited variability in the relatively small call graphs of PolyBench kernels, they are expected to become more informative when the framework is extended to larger, multi-module benchmark suites such as MiBench or SPEC CPU.

4.2. GNN Models

Graph Neural Networks (GNNs) are a class of neural network architectures designed to operate directly on graph-structured data through a message-passing process. In each layer, a node aggregates feature vectors from its neighbors, optionally applies weighting or attention, and then combines this information with its own features to produce an updated representation. By stacking multiple layers, GNNs capture both local and higher-order dependencies in the graph. In this study, the program is modeled as a graph $G = (V, E)$, where each node $v \in \mathrm{V}$ corresponds to a function in the application, and each edge $e \in \mathrm{E}$ represents a function call relationship. Node attributes are the static code features extracted from LLVM IR, while edges are derived from the function call graph. This formulation enables the models to learn jointly from the intrinsic properties of individual functions and the structural dependencies between them.

Table 3. *LLVM features selected in our study*

Feature Name	Definition
fpOps	Number of floating-point operations (e.g., addition, subtraction, multiplication, division) executed by the function.
branches	Number of branch instructions (conditional and unconditional jumps) in the function.
loads	Number of load instructions that read data from memory.
stores	Number of store instructions that write data to memory.
callCount	Number of function calls made by the function.
stackAllocs	Number of stack memory allocations (e.g., alloca instructions) in the function.
globalAccesses	Number of accesses to global variables in the function.
loopCount	Number of loops identified in the function.
numArgs	Number of input arguments the function accepts.
instrCount	Total number of instructions in the function.
basicBlocks	Number of basic blocks in the function, measuring its control flow granularity.
comparisons	Number of comparison operations (e.g., icmp, fcmp) in the function.

The general message-passing framework of a GNN layer can be expressed as follows:

$$h_v^{(l+1)} = \sigma\left(\mathrm{COMBINE}^{(l)}\left(h_v^{(l)}, \mathrm{AGGREGATE}^{(l)}\left(\{h_u^{(l)} : u \in \mathcal{N}(v)\}\right)\right)\right) \tag{1}$$

where $h_v^{(l)}$ denotes the feature vector of node v at layer l, $\mathcal{N}(v)$ is the set of neighbors of v, $\mathrm{AGGREGATE}^{(l)}(\cdot)$ is a function that collects information from neighbors, $\mathrm{COMBINE}^{(l)}(\cdot)$ merges the aggregated neighbor features with the node's own features, and σ is a non-linear activation function, implemented as Rectified Linear Unit (ReLU), $\mathrm{ReLU(x)} = \max(0, \mathrm{x})$ in this work.

We evaluate three GNN architectures for function-level soft error vulnerability and criticality prediction: Graph Convolutional Network (GCN) (Kipf & Welling, 2017), Graph Attention Network (GAT) (Veličković et al., 2018), and GraphSAGE (Hamilton et al., 2017).

GCN operates by aggregating and averaging feature information from a node's neighbors, normalized by node degrees:

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

where $\hat{A} = A + I$ is the adjacency matrix with self-loops, $\widehat{D}$ is its degree matrix, $H^{(l)}$ is the feature matrix at layer l, $W^{(l)}$ is the trainable weight matrix, and σ is the activation function. This operation averages and propagates features across function call connections in the program graph.

GAT introduces an attention mechanism to assign importance weights to neighbors before aggregation:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}\, W^{(l)}h_j^{(l)}\right) \tag{3}$$

where α_{ij} is the learned attention coefficient between nodes i and j, computed via a softmax over the neighbors of i. This formulation allows the model to focus more on critical caller–callee relationships that may influence fault propagation.

GraphSAGE employs a neighborhood sampling strategy and concatenates the aggregated neighbor features with the node's own features:

$$h_i^{(l+1)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_i^{(l)}, \text{AGGREGATE}\left(\{h_j^{(l)}, \forall j \in \mathcal{N}(i)\}\right)\right)\right) \tag{4}$$

where AGGREGATE(·) is implemented as mean aggregation in this work. CONCAT preserves the node's own features alongside its neighborhood context, helping retain function-specific characteristics while incorporating surrounding structural information relevant to soft error vulnerability and criticality.

4.3. Problem Formulation

In this study, the prediction of function-level resilience characteristics is addressed as both a regression and a classification problem. The target variables are the SDC vulnerability score and the criticality score, which quantify the susceptibility of a function to silent data corruptions and its relative importance in the program, respectively. Since these scores are obtained as continuous values from fault injection experiments, they can naturally be modeled as regression targets. However, in many practical scenarios, it is also desirable to categorize functions into discrete classes (e.g., vulnerable or non-vulnerable) based on predefined thresholds. Therefore, we consider both formulations to provide a comprehensive analysis.

For the regression formulation, the goal is to predict the continuous SDC vulnerability and criticality scores for each function using static code features and graph-structured program representations. The regression models are trained to minimize the prediction error between the predicted and ground-truth scores. Model performance is evaluated using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE), which respectively capture the average magnitude of prediction errors and penalize larger deviations more heavily.

For the classification formulation, continuous target scores are converted into binary class labels according to predefined thresholds. A function is labeled as vulnerable if its SDC vulnerability score exceeds the vulnerability threshold (34% in Table 1), or as critical if its criticality score exceeds the criticality threshold (0.037 in Table 2). These thresholds were selected to maintain a balanced distribution of instances across the two classes. Under this setup, 42 of the 90 analyzed functions are labeled as vulnerable in the vulnerability classification task, and 43 of the 90 functions are labeled as critical in the criticality classification task. The classification models are trained to correctly assign each function to its respective class, with performance evaluated using accuracy, precision, recall, and F1 score. This dual problem formulation enables the analysis of both the exact magnitude of resilience metrics and their practical categorization into actionable classes.

4.4. Training and Evaluation

All GNN models are implemented in Python using the PyTorch Geometric library (Fey & Lenssen, 2019; Paszke et al., 2019), which provides optimized layers for GCN, GAT, and GraphSAGE. Training and evaluation are conducted on the constructed function-level call graphs with the extracted LLVM features as node attributes. We standardized the features using the StandardScaler, which transforms each feature to have zero mean and unit variance. This normalization ensures that all features contribute equally to the model regardless of their original scale.

We evaluated all GNN models using Leave-One-Out Cross-Validation (LOOCV) at the application level, where in each fold all functions from one PolyBench application were held out for testing and the remaining applications were used for training. This setup ensures that the models are tested on completely unseen applications, providing a robust assessment of generalization across different program structures. LOOCV was chosen because the dataset contains a limited number of applications, and this approach maximizes the amount of training data in each fold while still providing a strict separation between training and testing. For each run, we trained GCN, GAT, and GraphSAGE models with two message-passing layers followed by an output layer, using the Adam optimizer with default momentum coefficients ($\beta_1 = 0.9$, $\beta_2 = 0.999$), no weight decay and a baseline learning rate of 0.01 for 100 epochs. The hidden layer size is set to 8 units, and dropout was applied between layers to reduce overfitting. These are the baseline hyperparameters. We later perform hyperparameter tuning on the best-performing model to further optimize its performance. To ensure robustness, each experiment is repeated for 10 runs with different random seeds, and results are reported as the mean and standard deviation across all folds and runs.

Each PolyBench application is represented as an individual graph, with nodes corresponding to functions and directed edges representing static caller–callee relations. During leave-one-application-out cross-validation, one entire graph is held out for testing, while the remaining graphs form the training set. This setup ensures inductive generalization across unseen applications rather than transductive learning within a single disjoint graph.

5. RESULTS AND DISCUSSION

5.1. Regression Results

We evaluated the performance of GCN, GAT, and GraphSAGE models on SDC rate and criticality prediction using Mean Absolute Error (MAE) and Root-Mean Squared Error (RMSE) as primary error metrics. Table 4 shows the results of SDC prediction, whereas Table 5 shows the results of criticality prediction. Across 10 runs, all models achieved low MAE (0.14 – 0.16 for the SDC prediction and 0.17 - 0.20 for the criticality prediction) and RMSE (0.17 – 0.19 for the SDC prediction and 0.21 - 0.23 for the criticality prediction), indicating that their absolute prediction errors were small relative to the target scale.

Among models, GraphSAGE shows the lowest MAE and RMSE values for both cases. This indicates that aggregating neighborhood information via mean and concatenation (as GraphSAGE does) works better for our graph structure than GCN's averaging or GAT's attention. On the other hand, GAT improves both MAE and RMSE compared to GCN shows that the attention mechanism captures important neighbor relationships, but gains are smaller than expected, possibly due to limited graph complexity and small dataset.

Table 4. *Regression results of SDC prediction*

Model	**MAE**	**RMSE**
GCN	0.161 ± 0.010	0.191 ± 0.012
GAT	0.142 ± 0.004	0.167 ± 0.005
GraphSAGE	0.140 ± 0.003	0.167 ± 0.003

Table 5. *Regression results of criticality prediction*

Model	**MAE**	**RMSE**
GCN	0.201 ± 0.006	0.233 ± 0.008
GAT	0.179 ± 0.005	0.218 ± 0.005
GraphSAGE	0.173 ± 0.003	0.209 ± 0.004

5.2. Classification Results

In this section, we reformulate the problem as a binary classification task by labeling each function as vulnerable (if SDC rate > threshold) or critical (if criticality > threshold). We apply the same models used in the regression task and evaluate their performance using *accuracy*, *precision*, *recall*, and *F1 score* as comparison metrics. *Accuracy* measures overall correctness, *precision* reflects the proportion of correctly identified positive instances, *recall* indicates the proportion of actual positives correctly identified, and *F1 score* balances precision and recall by taking the harmonic mean of both.

Table 6 shows the classification results of the vulnerability prediction, whereas Table 7 shows the classification results of the criticality prediction. As in the regression task, GraphSAGE consistently provides the highest overall performance, achieving an F1 score of 0.68, outperforming GCN (0.65) and GAT (0.67) in the SDC vulnerability label prediction.

When the label is defined as criticality, all models show a noticeable performance increase. This indicates that incorporating execution time creates a more separable and meaningful classification. GraphSAGE again achieves the best results (F1 = 0.80), slightly ahead of GCN (0.78) and notably stronger than GAT (0.74).

Table 6. *Classification results of SDC vulnerability label prediction*

Metric	GCN (±)	GAT (±)	GraphSAGE (±)
Accuracy	0.63 ± 0.02	0.63 ± 0.02	0.68 ± 0.02
Precision	0.61 ± 0.03	0.62 ± 0.02	0.69 ± 0.03
Recall	0.80 ± 0.03	0.84 ± 0.03	0.77 ± 0.05
F1 Score	0.65 ± 0.03	0.67 ± 0.02	0.68 ± 0.04

Table 7. *Classification results of criticality label prediction*

Metric	GCN (±)	GAT (±)	GraphSAGE (±)
Accuracy	0.83 ± 0.02	0.78 ± 0.03	0.83 ± 0.01
Precision	0.78 ± 0.03	0.74 ± 0.05	0.82 ± 0.01
Recall	0.83 ± 0.03	0.80 ± 0.04	0.85 ± 0.01
F1 Score	0.78 ± 0.03	0.74 ± 0.04	0.80 ± 0.01

Furthermore, we performed hyperparameter tuning for the GraphSAGE model for both SDC vulnerability and criticality label prediction tasks. To ensure unbiased performance estimation and robust hyperparameter selection, we adopt a nested cross-validation (CV) strategy. The outer loop follows a leave-one-application-out (LOOCV) protocol: in each fold, all functions from one PolyBench application are held out as the test set, while the remaining applications form the training pool. Inside each outer fold, we perform an inner CV loop (also LOOCV across the training applications) to tune key hyperparameters such as hidden units {8, 16, 32} , learning rate {0.01, 0.005}, number of layers {2, 3}, and dropout rate {0.2, 0.4} based solely on validation performance. The optimal configuration determined by this inner search is then retrained on the full outer-training data and evaluated once on the held-out application. This procedure prevents information leakage from the test set into the model selection process and yields an unbiased estimate of generalization ability.

Additionally, we implement fold-wise feature scaling to further eliminate leakage risks. In every fold, the StandardScaler is fit exclusively on the training data within that fold and subsequently applied to transform the validation and test data. This ensures that statistical information from the test set (such as feature means and variances) is never exposed to the model during training or preprocessing. Together, nested CV and fold-wise scaling provide a rigorous evaluation setup that closely reflects the model's expected performance on unseen applications.

Overall, the nested leave-one-application-out evaluation conducted across 30 applications resulted in an average F1 score of 0.68 (± 0.03) for predicting the SDC vulnerability level and 0.79 (± 0.02) for predicting the criticality level. This minor reduction reflects the removal of optimistic bias introduced by evaluating tuned models on the same data, indicating that the nested CV provides a more conservative and reliable estimate of generalization performance.

5.3. Feature Selection

In this section, we focus on the importance of the features for the classification task and investigate the Pearson and Spearman correlation of the features with the corresponding classes. Pearson measures the strength of a linear relationship between two variables, while Spearman measures the strength of a monotonic (rank-based) relationship, regardless of linearity. Pearson and Spearman correlation results for the SDC vulnerability and criticality label prediction are shown in Figure 2 and 3, respectively. Here, darker red color represents strong positive correlations and darker blue color indicates negative correlations. Our first observation from these results is that, while a subset of features shows both positive and negative correlations with SDC vulnerability prediction, a larger number of features are highly correlated with criticality prediction.

We identify top-5 features (a common approach in scientific studies) that are highly correlated based on Spearman correlation since it can capture non-linear relations for complex structures. For the SDC vulnerability label prediction, the top-5 features identified are: *callCount*, *stackAllocs*, *globalAccesses*, *fpOps*, and *numArgs*. For the criticality label prediction, the top-5 features are: *fpOps*, *stores*, *loopCount*, *loads* and *instrCount*. The selected features differ between the two tasks, with the exception of *fpOps*, which appears in both.

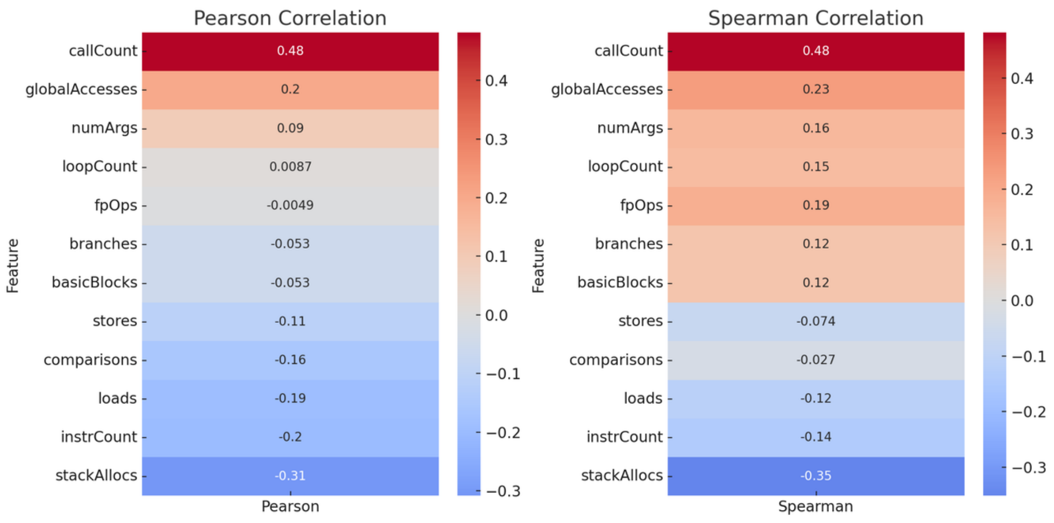

Figure 2. *Feature correlation results based on Pearson and Spearman correlation in vulnerability label prediction*

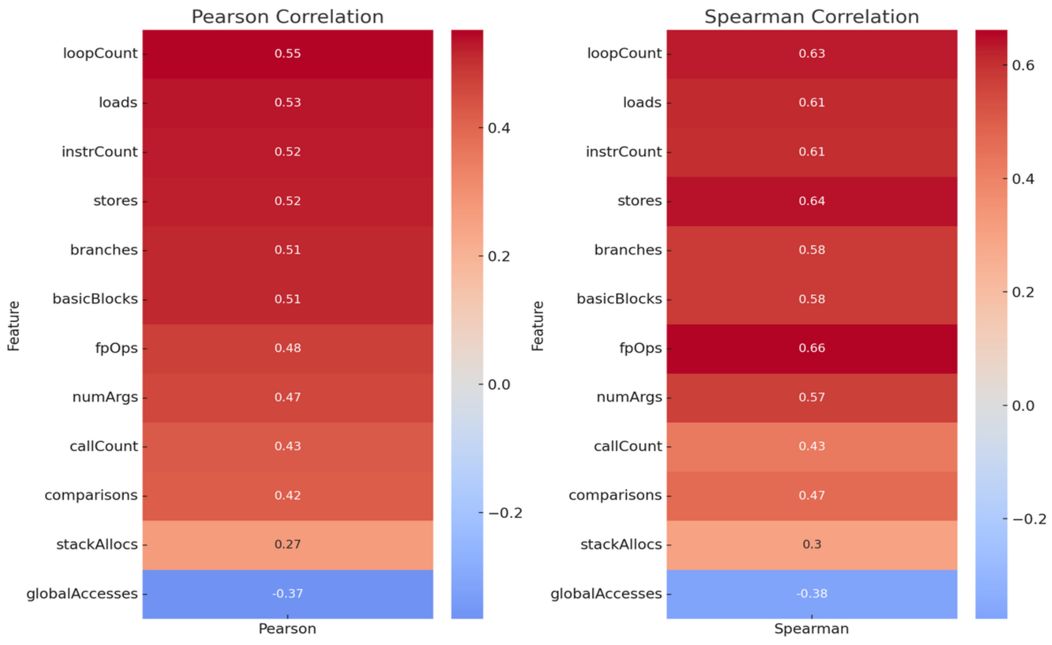

Figure 3. *Feature correlation results based on Pearson and Spearman correlation in criticality label prediction*

We re-evaluated the GNN models using only these top-5 features, with the results presented in Table 8 for SDC vulnerability prediction and Table 9 for criticality prediction, respectively. While the GCN model exhibited a significant drop in accuracy, GraphSAGE maintained stable or slightly degraded performance, indicating its robustness to feature reduction. These results suggest that using the full set of features is beneficial for achieving maximum accuracy.

Table 8. *Classification results of SDC label prediction with top-5 selected features based on Spearman correlation*

Metric	GCN	GAT	GraphSAGE
Accuracy	0.54 ± 0.03	0.60 ± 0.06	0.66 ± 0.03
Precision	0.56 ± 0.04	0.60 ± 0.05	0.69 ± 0.02
Recall	0.75 ± 0.05	0.74 ± 0.07	0.76 ± 0.03
F1 Score	0.58 ± 0.04	0.61 ± 0.06	0.67 ± 0.02

Table 9. *Classification results of criticality label prediction with top-5 selected features based on Spearman correlation*

Metric	GCN (±)	GAT (±)	GraphSAGE (±)
Accuracy	0.72 ± 0.03	0.77 ± 0.02	0.82 ± 0.01
Precision	0.75 ± 0.04	0.76 ± 0.04	0.81 ± 0.02
Recall	0.68 ± 0.03	0.77 ± 0.03	0.84 ± 0.02
F1 Score	0.67 ± 0.03	0.73 ± 0.03	0.79 ± 0.02

In addition to conventional correlation analysis (e.g., Pearson and Spearman), we evaluated feature relevance using a model-based permutation importance metric (ΔF1). Unlike correlation coefficients, which quantify only linear (Pearson) or monotonic (Spearman) associations between individual features and target labels, the permutation-based approach captures each feature's marginal contribution to the trained model's predictive performance. For each feature, its values are randomly permuted across all samples, breaking its association with the target, while all other features remain fixed, and the model is re-evaluated on the perturbed data. The resulting decrease in F1 score (ΔF1) quantifies the feature's contribution to correct classification, with larger ΔF1 values indicating higher importance.

We determine the top-5 features identified by permutation ΔF1 importance and evaluate the model's performance for both criticality and SDC vulnerability prediction, shown in Table 10. Remarkably, the criticality classification task preserved nearly the same performance (F1 ≈ 0.79–0.80) compared to the full 12-feature models, indicating that the essential predictive signal is captured by a small subset of memory-intensity metrics (loads, stores, instrCount, globalAccesses, and fpOps). In contrast, SDC vulnerability prediction exhibited a modest performance decrease (F1 ≈ 0.65–0.67), suggesting that multiple control-flow and stack-related features jointly contribute to fault propagation behavior (callCount, stackAllocs, branches, comparisons, basicBlocks).

5.4. Discussion

The experimental results demonstrate the effectiveness of Graph Neural Networks (GNNs) in predicting both soft error vulnerability and function criticality in scientific applications. Among the evaluated models, GraphSAGE consistently outperformed GCN and GAT across both regression and classification tasks. This suggests that GraphSAGE's neighborhood aggregation strategy, which combines mean aggregation with concatenation, better captures the structural information in our function level call graphs compared to GCN's averaging or GAT's attention mechanism.

The regression results indicate low prediction errors for all models, with GraphSAGE achieving the lowest MAE and RMSE in both SDC and criticality prediction. The slightly better performance of GAT over GCN also confirms that attention mechanisms can enhance performance, though the gains were modest, possibly due to the relatively small graph size and limited dataset complexity.

In the classification task, models performed better when the label was defined based on criticality (which combines SDC rate and execution time) rather than SDC rate alone. This implies that incorporating execution time leads to a more informative and discriminative label, allowing models to more easily separate classes. Again, GraphSAGE achieved the highest F1 scores, reaching 0.80 for criticality prediction.

The feature selection analysis further validated the impact of input features. While reducing features to the top-5 set caused performance degradation for GCN and GAT, GraphSAGE remained relatively robust, maintaining comparable classification accuracy. This highlights its ability to effectively learn from limited but highly informative features, while also showing that full feature sets still yield optimal performance. Additionally, the results obtained using only the top-five features indicate that the criticality label serves as a more stable and discriminative learning target than the binary SDC vulnerability label. Since criticality integrates both fault manifestation likelihood and execution-time weighting, it produces smoother decision boundaries and reduces noise caused by binary thresholding of SDC rates, enabling GNNs to generalize more effectively across applications.

Table 10. *Classification performance of GNN models using only the top-5 features ranked by permutation ΔF1 importance*

Model	**Label Type**	**Accuracy (±)**	**Precision (±)**	**Recall (±)**	**F1 (±)**	**Top-5 Features (by permutation ΔF1)**
GraphSAGE	Criticality	0.81 ± 0.01	0.80 ± 0.02	0.84 ± 0.01	**0.79 ± 0.01**	*loads, globalAccesses, instrCount, stores, fpOps*
GraphSAGE	SDC Vulnerability	0.67 ± 0.03	0.66 ± 0.02	0.77 ± 0.05	**0.66 ± 0.03**	*callCount, stackAllocs, comparisons, fpOps, basicBlocks*
GCN	Criticality	0.81 ± 0.01	0.76 ± 0.02	0.88 ± 0.02	**0.79 ± 0.02**	*callCount, globalAccesses, stores, fpOps, instrCount*
GCN	SDC Vulnerability	0.62 ± 0.01	0.60 ± 0.01	0.80 ± 0.02	**0.65 ± 0.01**	*basicBlocks, branches, comparisons, loopCount, numArgs*
GAT	Criticality	0.82 ± 0.02	0.83 ± 0.02	0.83 ± 0.02	**0.80 ± 0.02**	*globalAccesses, callCount, loads, stores, instrCount*
GAT	SDC Vulnerability	0.65 ± 0.01	0.63 ± 0.02	0.82 ± 0.03	**0.67 ± 0.02**	*comparisons, branches, basicBlocks, loads, loopCount*

While this study evaluates the proposed framework on the PolyBench benchmark suite, which is widely used for compiler and resilience research, it primarily represents compute-intensive CPU workloads. Although these applications capture diverse numerical and memory access behaviors, they may not fully reflect other software domains, such as irregular or data-intensive workloads. Future work will therefore focus on extending the framework to additional benchmark suites such as MiBench and SPEC CPU to assess generalizability and scalability across heterogeneous application classes.

6. CONCLUSION

This paper presented a function-level soft error vulnerability and criticality prediction framework for CPU-based scientific applications, leveraging LLVM-based static feature extraction and graph neural network modeling. By constructing function call graphs enriched with 12 static code features, the proposed method enables GCN, GAT, and GraphSAGE models to learn both functional properties and structural dependencies relevant to SDC propagation. The dual formulation as regression and classification tasks allows for both precise vulnerability/criticality prediction and actionable binary classification. Experimental results on PolyBench benchmarks show that GraphSAGE achieves the best overall performance, maintaining robustness even under feature reduction. Compared to traditional fault injection methods, our approach offers a more practical solution for fine-grained resilience analysis. Future work will focus on scaling the framework to larger and more diverse workloads, integrating dynamic execution features such as runtime instruction mix and memory-access intensity, and exploring Program Dependence Graph (PDG)–based representations to capture deeper data and control dependencies that influence fault propagation. In addition, alternative graph-based and hybrid learning architectures beyond standard GNNs will be investigated to further improve predictive generalization.

CONFLICT OF INTEREST

The author declares no conflict of interest.

REFERENCES

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys, 51(4)*. https://doi.org/10.1145/3212695

Arslan, S., & Unsal, O. (2021). Efficient selective replication of critical code regions for SDC mitigation leveraging redundant multithreading. *Journal of Supercomputing, 77(12),* 14130–14160. https://doi.org/10.1007/s11227-021-03804-6

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., & Tao, C. (2022, May 21-29). *MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks.* In: Proceedings of the 44th International Conference on Software Engineering (ICSE'22) (pp. 1456–1468), Pittsburgh Pennsylvania. https://doi.org/10.1145/3510003.3510219

Fey, M., & Lenssen, J. E. (2019). *Fast graph representation learning with PyTorch Geometric.* In: Proceedings of the ICLR Workshop on Representation Learning on Graphs and Manifolds. https://doi.org/10.48550/arXiv.1903.02428

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). *Inductive representation learning on large graphs*. In: Advances in Neural Information Processing Systems (NeurIPS), (pp. 1024–1034). https://doi.org/10.48550/arXiv.1706.02216

Kipf, T. N., & Welling, M. (2017). *Semi-supervised classification with graph convolutional networks*. In: International Conference on Learning Representations (ICLR). https://doi.org/10.48550/arXiv.1609.02907

Laguna, I., Schulz, M., Richards, D. F., Calhoun, J., & Olson, L. N. (2016, March 12-18). *IPAS: Intelligent protection against silent output corruption in scientific applications*. In: Proceedings of the 2016

International Symposium on Code Generation and Optimization (CGO '16). Association for Computing Machinery (pp. 227–2389, Barcelona, Spain. https://doi.org/10.1145/2854038.2854059

Lu, Q., Farahani, M., Wei, J., & Pattabiraman, K. (2015, August 3-5). *LLFI: An intermediate code-level fault injection tool for hardware faults*. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN '15), Vancouver, BC, Canada. https://doi.org/10.1109/QRS.2015.13

Mukherjee, S. S., Kontz, C. T., & Reinhardt, S. K. (2002, May 25-29). *Detailed design and evaluation of redundant multithreading alternatives*. In: Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA), (pp. 99–110), Anchorage, AK, USA. https://doi.org/10.1109/ISCA.2002.1003566

Ni, C., Guo, X., Zhu, Y., Xu, X., & Yang, X. (2024, September 11-15). *Function-level Vulnerability Detection Through Fusing Multi-Modal Knowledge*. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23). IEEE Press, (pp. 1911–1918), Luxembourg, Luxembourg. https://doi.org/10.1109/ASE56229.2023.00084

Öz, I., and Arslan, S. (2021). Predicting the soft error vulnerability of parallel applications using machine learning. *International Journal of Parallel Programming, 49*, 410–439. https://doi.org/10.1007/s10766-021-00707-0

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). *PyTorch: An imperative style, high-performance deep learning library*. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), Advances in Neural Information Processing Systems 32 (NeurIPS 2019) (pp. 8024–8035). Curran Associates, Inc.

Pouchet, L.-N. (2012). Polybench/c: The polyhedral benchmark suite. (Accessed: August 8, 2025) https://www.cs.colostate.edu/~pouchet/software/polybench/

Topçu, B., & Öz, I. (2023). Soft error vulnerability prediction of gpgpu applications. *The Journal of Supercomputing, 79*, 6965–6990. https://doi.org/10.1007/s11227-022-04933-2

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Li`o, P., & Bengio, Y. (2018). *Graph attention networks*. In: International Conference on Learning Representations (ICLR). https://doi.org/10.48550/arXiv.1710.10903

Wei, X., Zhao, J., Jiang, N., & Yue, H. (2023, October 20–22). *GLAM-SERP: Building a graph learning-assisted model for soft error resilience prediction in GPGPUs*. In: Algorithms and Architectures for Parallel Processing: 23rd International Conference, ICA3PP 2023, Proceedings, Part IV, LNCS 14490, (pp. 419–435), Tianjin, China. https://doi.org/10.1007/978-981-97-0859-8_25

Zou, D., Wang, S., Xu, S., Li, Z., & Jin, H. (2021). µVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing*, *vol. 18, no. 05*, pp. 2224-2236, Sept.-Oct. 2021, https://doi.org/10.1109/TDSC.2019.2942930