GLAIVE: Graph Learning Assisted Instruction Vulnerability Estimation

Jiajia Jiao[12*], Debjit Pal[1*], Chenhui Deng[1], Zhiru Zhang[1]
[1]Cornell University, Ithaca NY 14853, USA; [2]Shanghai Maritime University, China.
Email: {jiaojiajia@shmtu.edu.cn}, {debjit.pal, cd574, zhiruz}@cornell.edu

***Abstract*—Due to the continuous technology scaling and lowering of operating voltages, modern computer systems are highly vulnerable to soft errors induced by the high-energy particles. Soft errors can corrupt program outputs leading to silent data corruption or a Crash. To protect computer systems against such failures, architects need to precisely and quickly identify vulnerable program instructions that need to be protected. Traditional techniques for program reliability estimation either use expensive and time-consuming fault injection or inaccurate analytical models to identify the program instructions that need to be protected against soft errors. In this work, we present GLAIVE, a graph learning-assisted model for fast, accurate, and transferable soft-error induced instruction vulnerability estimation. GLAIVE leverages a synergy between static analysis and data-driven statistical reasoning to automatically learn signatures of instruction-level vulnerabilities and their propagation to program outputs using a fine-grain error propagation information from the bit-level program graphs of a set of realistic benchmarks. Our experiments show that the learned knowledge of instruction vulnerability is transferable to unseen programs. We further show that GLAIVE can achieve an average 221× speedup and up to 33.09% lower program vulnerability estimation error as compared to a baseline fault-injection technique, up to 30.29% higher vulnerability estimation accuracy, and on average can cover up to 90.23% vulnerable instructions for a given protection budget compared to a set of baseline machine learning algorithms.**

I. Introduction

Progressive technology scaling and lowering of operating voltages have made contemporary and future computer systems more susceptible to high-energy particles induced soft errors (*i.e.*, transient hardware faults). The soft errors propagate following the program execution flow and corrupt program output leading to silent data corruptions (SDCs) or a System Crash. Hardware-only protection solutions are becoming increasingly infeasible for on-field deployment due to the on-chip power and area constraints [4]. Consequently, there is an urgent need to develop fault-tolerant programs by quickly and accurately estimating the soft-error induced program vulnerability.

Traditionally, soft error estimation approaches primarily focus on fault injection (FI) campaigns and analytical models. FI [10], [12] involves manipulating program states by randomly injecting one fault per campaign to emulate a transient hardware fault. The faulty program is allowed to completion to determine if the injected fault has been *Masked* or caused an *SDC* or a *Crash*. However, a realistic program contains billions of instructions necessitating a large number of *time-consuming* FI campaigns to achieve statistically significant results. Alternatively, researchers have developed analytical models [5] for error propagation for fast identification of vulnerable instructions. However, the analytical models are *inaccurate* and suffer from *scalability*. Recently, machine learning-based (ML) methods [6], [9] are increasingly used to estimate instruction vulnerability. However, each of these methods suffers from high accuracy loss, small training datasets, and requires time-consuming retraining for unseen programs making them infeasible for realistic program vulnerability estimation. Consequently, it is crucial than ever before that we address the current lack of scalable, fast, accurate, and transferable solutions for instruction vulnerability estimation for realistic applications.

In this work, we develop GLAIVE, a graph learning assisted model to automatically learn bit-level vulnerabilities and their propagation patterns to program outputs to compute individual instruction vulnerability. We find two key factors that determine the soft error propagation to program outputs – i) the bit location of a fault occurrence and ii) the instructions in the control-data flow path from the fault occurrence location to the program output. Both of these are local structures around program instructions and since graph learning can learn neighborhood information from graph-structured data, we use graph learning to assist instruction vulnerability estimation. GLAIVE learns fine-grained bit-level error propagation patterns by using program control-data flow graphs (CDFG) and data-driven statistical reasoning such as graph learning [7] in a complementary and synergistic way resulting in a model that is *scalable to large programs containing millions of instructions, can quickly and accurately estimate instruction vulnerability, and can be transferred to unseen programs without retraining*. Our primary contributions in this work are as follows:

- To the best of our knowledge, GLAIVE is the first work for instruction vulnerability estimation that automatically learns fine-grained bit-level error propagation patterns using a synergy of static and data-driven graph learning-based analysis.
- We show with empirical evidence that the bit-level information is significantly more useful than word-level information in learning instruction vulnerability and its propagation patterns to program outputs.
- We demonstrate that GLAIVE achieves up to average 221× speedup with average 90.23% coverage of most vulnerable instructions under varying protection budgets over state-of-the-art FI [12], and up to 30.29% higher accuracy and on average 33.09% lower program vulnerability error than a set of baseline ML methods.

[*]Equal contribution; Work done while Jiajia Jiao was visiting Cornell.

II. Preliminaries

A. Fault model

In this work, we consider transient hardware faults due to single-bit upset in the computational elements of a processor, *e.g.*, registers, caused by high-energy particles and/or random noise in circuits. Single-bit upset during application execution can cause SDC or can crash the program. We do not consider memories and caches as they are usually protected by error correction code (ECC) or parity bits in modern designs. Further, we assume that the processor's control logic is protected against faults via control-flow checking. However, the program may take a faulty legal branch (a legal execution path caused via a wrongly taken branch due to faults propagating to it). Specifically, we consider faults that happen in the registers that store instruction inputs and outputs. Our fault model is aligned with other state-of-the-art methods [8], [9], [12].

B. Terminologies

Bit vulnerability: The impact of each of the single-bit upsets is defined as bit vulnerability. Bit vulnerability are of three types – *Crash*, *SDC*, and *Masked*.

Crash: The raising of an exception or hardware trap followed by program termination by the OS due to an illegal action (*e.g.*, out-of-bound memory access, divide-by-zero) of a program is defined as a crash.

SDC: An SDC is defined as a mismatch between the output of a faulty execution and an error-free execution of a program.

Masked: A fault during a program execution is said to be masked if the output of the execution matches to that of program's error-free execution.

Error propagation: Error propagation implies post-activation, a fault has affected one or more program states via control and/or data paths during program execution and caused the execution to *fail* via a Crash or an SDC.

Vulnerable instruction: An instruction is vulnerable if a bit-upset on that instruction leads to an SDC or a program crash. We assume a ranking ordering `Crash` $\rightarrow$ `SDC` $\rightarrow$ `Masked` to select most vulnerable instructions in a program.

Instruction vulnerability: The instruction vulnerability of an instruction I is defined as a tuple $I_v = \langle I_C, I_S, I_M \rangle$ where I_C, I_S, and I_M represents Crash probability, SDC probability, and Masked probability of an instruction, respectively. If an instruction I contains a total of N_U bit upsets that lead to N_C, N_S, and N_M numbers of Crashes, SDC, and Masked, respectively, then $I_C = N_C/N_U$, $I_S = N_S/N_U$, and $I_M = N_M/N_U$ where $I_C + I_S + I_M = 1$. Instruction vulnerability imposes a ranking $\mathcal{R}$ among program instructions. The **program vulnerability** P_v is defined as the weighted sum of all instruction vulnerabilities where the weight is faults injected in an instruction expressed as a fraction of total injected faults in a program. The **program vulnerability error** is the sum of absolute errors between estimated program vulnerability and FI for each of the Crash, SDC, and Masked.

Top-K vulnerable instruction set: Given a instruction ranking $\mathcal{R}$, we define top-K vulnerable instruction set S_K as the top-K instructions that need to be protected to ensure correct program execution under a given protection budget K. We set the size of S_K to $argmin(N \times K, N_v)$ where N=number of program instructions, N_v=number of vulnerable instructions.

Top-K coverage: The top-K coverage S_C^K is defined as the fraction of total vulnerable instructions that are estimated correctly by an arbitrary estimation approach *i.e.*, $S_C^K = |(S^* \cap S_K)|/|S_K|$ where S^* are the ideal top-K vulnerable instructions estimated accurately by fault injection and S_K estimated by proposed estimation approach.

C. Fault injection tool: gem5-Approxilyzer

gem5-Approxilyzer [12] is an efficient and accurate fault injection tool to characterize the impact of a transient hardware fault induced by a single-bit upset on a program execution. It systematically analyzes all potential error sites in a program to select a small subset of locations for error injections. gem5-Approxilyzer employs a combination of error-pruning techniques *e.g.*, predicting program outcome post fault injections, identifying equivalent classes of faults, to reduce the number of actual fault injections. In this work, we use gem5-Approxilyzer to generate large-scale ground truth fault injection data.

D. Graph neural networks (GNNs)

Recent years have seen a surge of interest in GNNs, which are also known as deep learning on graphs. Different from traditional graph learning techniques, GNNs are able to incorporate both graph topology and node attribute information, which makes GNNs to achieve state-of-the-art results in various graph-based tasks, including node classification, link prediction, and community detection [3], [15].

GNNs aim to encode graph structural information into low-dimensional vectors that can be utilized for downstream tasks. Specifically, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ and $\mathcal{E}$ represents node and edge set, respectively, let A be a $n \times n$ adjacency matrix where $n = |V|$, and X be the $n \times k$ node attribute matrix whose i^{th} row represents node attribute information of the i^{th} node in a k-dimension vector. A GNN model is essentially a trainable function F such that $Y = F(A, X)$ where Y is the output $n \times d$ embedding matrix, and Y_i represents embedding vector of i^{th} node in d dimension.

Although there are various GNN models, they can be broadly grouped into two categories – i) *transductive* and ii) *inductive*. A transductive model requires to see the entire graph structure during training to produce node embedding vectors, which implies that the model needs to be retrained when the graph structure changes. In contrast, an inductive model learns a general rule via learning an aggregation function, which collects attribute information from neighbors without knowing the whole graph structure. Thus, the trained inductive model can be directly applied to unseen graphs without retraining. In this work, we repurpose an inductive GNN model GraphSAGE [7] to predict top-K vulnerable instructions.

E. Problem formulation

We propose to automatically estimate the instruction vulnerability of each of the instructions of a program due to single-

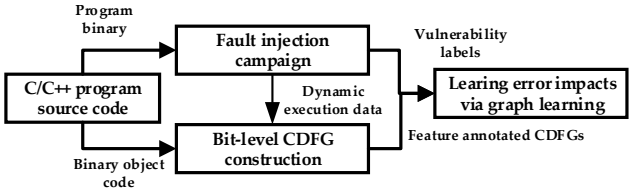


Fig. 1: **Overall GLAIVE workflow.**

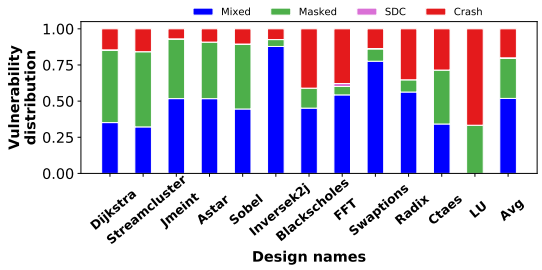


Fig. 2: **Vulnerability distributions of different benchmarks.**

bit upsets and induce a ranking to maximize top-K coverage using graph learning techniques. Specifically, given a program source code (in C/C++), a set of program inputs, instruction set architecture (ISA) of the target hardware, and a protection budget K, our method GLAIVE computes I_v for each of the instructions and induces a ranking $\mathcal{R}$ such that top-K coverage is maximized and program vulnerability error is minimized.

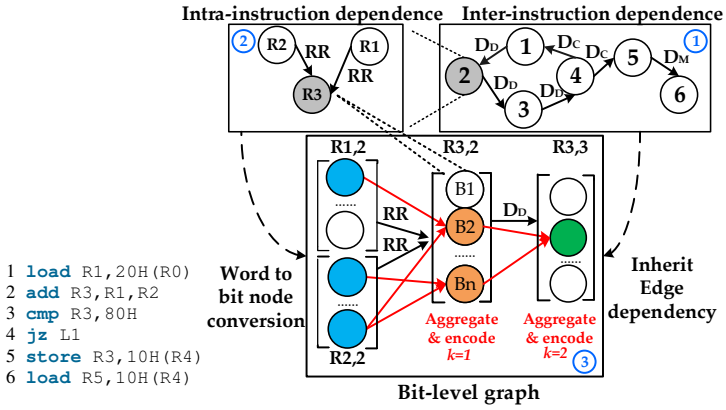
III. Proposed Approach: GLAIVE

A. Workflow and insights

We show our proposed workflow in Figure 1. GLAIVE[1] relies on two key insights – i) instruction vulnerabilities are dependent on the bit-level patterns of instructions and ii) failure of program execution due to error propagation is directly influenced by control and data dependencies that exist among various program instructions. Based on these correlations, we extract bit-level CDFG from program binaries and annotate each of the CDFGs with additional node features extracted from bit-level FI campaigns performed on program binaries using gem5-Approxilyzer. We use bit-level CDFG, annotated node features, and program execution status (*i.e.*, pass/fail) from the FI campaign to train our inductive GNN model. Post-training, we use our model to infer instruction vulnerabilities from bit-level CDFGs. Our technique can automatically learn instruction vulnerability and induce a ranking among program instructions quickly and accurately, thereby avoiding time-consuming FI campaigns and inaccuracy of analytical methods.

To support our key insights, we performed an initial study of vulnerability distributions on a set of benchmarks as shown in Figure 2. We find that up to 87.8% (on average 51.88%) instructions have mixed vulnerabilities of Masked, SDC, and Crash based on the bit location of soft error occurrence. This implies that program output is highly correlated with the bit location of a soft error rendering word-level information ineffective for instruction classification. In addition, the relative location of soft error occurrence (*e.g.*, MSB vs. LSB in a

[1]GLAIVE is available at https://github.com/cornell-zhang/GLAIVE.



(a) (b)

Fig. 3: **Instructions to bit-level CDFG construction** – (a) A machine language code snippet. (b) Bit-level CDFG for the code of Fig. 3a. D_D, D_C, D_M, RR: Data/Control/Memory/Register-register dependency. $R_{i,j}$: i^{th} register of the j^{th} instruction. B_i: i^{th} bit of a register. k: k^{th} iteration of aggregation.

control instruction) can change the overall program execution path resulting in a Crash due to an illegal operation. Therefore exploiting bit-level information is a natural choice for accurate instruction vulnerability estimation.

B. Bit-level graph features and ground truth extraction

In this step, we use a program source code (preferably in C/C++) and a suite of test vectors as inputs and produce a bit-level CDFG annotated with additional node features that can be used for graph learning in the next phase.

We compile a program source code to generate target architecture specific binary and perform static and dynamic analysis on the binary. For *static analysis* we use `objdump` to disassemble the binary in assembly instructions. We apply static analysis on the assembly code to generate instruction-level CDFG where each node in the CDFG is an instruction and each directed edge represents a inter-instruction control (D_C) / data (D_D) / memory (D_M) dependency among a pair of instructions. In Figure 3b we show the instruction-level CDFG annotated with instruction line number (marked as ①) for the assembly code of Figure 3a. To capture the intra-instruction dependency patterns of instruction operands, each instruction node is replaced with a operand-level graph where each node in the operand-level graph is an instruction operand and each directed edge represents a data dependence. For example, in Figure 3b we replace node 2 with R_1, R_2, and R_3, the operands of `add` instruction in line 2 (marked as ②). Although the edges in the instruction-level graph can capture error propagation across instructions, it fails to capture bit-level error propagation at the operand level. Hence each word-level operand in the operand-level graph is bit-blasted and a directed edge is added from each of the bit-level nodes of a source operand to each of the bit-level nodes of a destination operand. For example, in Figure 3b, we bit blast register R_3 to its constituent n bits (B_1, …, B_n) (marked as ③).

TABLE I: **CDFG node features – B, I**: Bit/instruction-level feature. §: Register is characterized based on its data content. ¶: Auxiliary features only used for pre and post-processing of graphs. **NA**: Not applicable.

Description	Type		Example	Representation
	B	I		
Op code	•	•	`ADD/SUB`	Boolean vector
Op code type	•	•	`Control`	Boolean
			`Memory related`	
Register name	•		R_3	Boolean vector
Bit location	•		15	
Register type§	•		`int`	Boolean
Register location	•		`src`	
Static PC¶	•		`0x401a88`	NA
Dynamic instance¶	•		22504439	
			7103000	

In order to effectively learn bit-level local patterns of instruction operands and its correlation with error propagation our graph learning framework needs to learn both structural and contextual information. The structural information, *i.e.*, node connectivities is directly available from the graph encoded as an adjacency matrix. To learn contextual information, *i.e.*, local neighborhood around each of the nodes of the bit-level CDFG, we use additional node features including `opcode`, `opcode type`, `source/destination register name`, `register location`, and `register type`, and `bit location`. We have shown example of each of the node features in Table I.

To supervise embedding generation during graph learning, we need ground truth, *i.e.*, whether a bit-level fault can cause a Crash/SDC or is Masked in a program execution. To extract ground truth, we perform bit-level FI campaign using the program binary. We use annotated bit-level CDFG and ground truth in the next phase for graph learning.

C. Graph neural network model

Instruction vulnerability depends on a plethora of node attributes and local structures of a bit-level CDFG such as the instruction type, bit location of fault occurrence in the operand registers of an instruction, fault's proximity to control/data paths to propagate to program output to cause either a Crash or an SDC. Since GNNs can incorporate both local structures and node attributes in node embedding vectors, we leverage GNNs to estimate instruction vulnerability. We formulate our task as a bit node classification problem. Specifically, given a bit-level CDFG, we label a node 0 if a fault at the node is Masked, 1 if a fault at the node causes an SDC, 2 if a fault at the node causes a Crash. We apply GNN model to this ternary node classification task.

We repurpose GraphSAGE [7], a recent GNN model, to learn node embedding inductively. Given a graph $\mathcal{G} = (V, E)$, GraphSAGE learns an aggregation function ($AGG_k(v)$) for node v by collecting information from all neighbors $u \in \mathcal{N}(v)$ and produces node embedding h_v^k in the k^{th}-iteration as follow.

$$h_v^k = \sigma(\mathbf{W}^k.CON(h_v^{k-1}, AGG_k(h_u^{k-1}, \forall u \in \mathcal{N}(v)) \quad (1)$$

where $\mathcal{N}(v)$ consists of 1-hop neighbors of node v, CON is a feature-wise concatenation function, $\mathbf{W}^k$ is a learnable weight matrix at the k^{th}-iteration. At $k = 0$, $h_v^0 = x_v$, where x_v is the initial attribute vector for node v.

TABLE II: **Details of experimental benchmark with dataset split**. **C**: Control-sensitive benchmarks. **D**: Data-sensitive benchmarks. **BL**: Bit-level datapoints. **IL**: Instruction-level datapoints. **TT**: Training/Testing dataset. **V**: Validation dataset.

Category	Type/Name	Datapoints BL (In thousands)	Datapoints IL (In thousands)	Benchmark description Suite	Benchmark description Domain	Dataset split
C	Dijkstra	643	2.552	Others	Path search	TT
	Astar	1214	0.677			
	Streamcluster	605	2.801	Parsec	Computer vision	
	Jmeint	234	0.866	AXbench	Robotics	
	Sobel	864	3.465		Image processing	
	Inversek2j	359	0.943		3-D gaming	V
D	Blackscholes	109	0.217	Parsec	Finance	TT
	Swaptions	632	1.844			
	FFT	632	1.238	Splash	Signal processing	
	Radix	296	1.121		Sorting	
	Ctaes	232	0.828	Others	Bitcoin core	
	LU	0.192	0.003	Splash	Computing	V

Augmented GNN model for bit vulnerability learning: In bit-level CDFG, the error propagates from one or more previous instructions to later instructions. This requires a node embedding to primarily aggregate information from preceding nodes in its neighborhood. The AGG function of vanilla GraphSAGE considers all neighbor nodes uniformly without explicitly considering predecessors, thereby considerably limiting its ability to learn accurate embedding for instruction vulnerabilities. To address this issue, we augment the vanilla GraphSAGE to collect and aggregate information from predecessor neighbors of a node and use it to update node embedding at the k^{th}-iteration. We augment Eq. (1) as follows.

$$h_v^k = \sigma(\mathbf{W}^k.CON(h_v^{k-1}, AGG_k^{\mathcal{PR}}(h_u^{k-1}, \forall u \in \mathcal{PR}(v)) \quad (2)$$

where $\mathcal{PR}(v)$ represents the set of predecessor nodes of node v. We use a $MEAN$ aggregator of Eq. (3) as suggested in [7].

$$AGG_k^{\mathcal{PR}}(h_u^{k-1}, \forall u \in \mathcal{PR}(v)) = \frac{1}{|\mathcal{PR}(v)|} \sum_{i \in \mathcal{PR}(v)} h_i^{k-1} \quad (3)$$

This modification captures directionality of error propagation and improves instruction vulnerability estimation accuracy. For example, in Figure 3b at $k = 1$, to compute node embedding of Orange nodes, we aggregate and encode information from its predecessors, *i.e.*, Blue nodes (marked as ③).

D. Learning instruction and program vulnerabilities

GLAIVE uses bit vulnerability distribution from the augmented GNN model to compute instruction vulnerability and program vulnerability as defined in Section II-B. We use instruction vulnerability ranking to select top-K vulnerable instructions for a given protection budget K.

IV. Experimental setup

Benchmarks: We use 10 benchmarks to construct our training and testing datasets, as well as 2 benchmarks for validation datasets from Parsec, Splash-2 [2], and AXBench [13] as shown in Table II. The benchmarks are classified into two categories: data-sensitive and control-sensitive. For each category of six benchmarks, we train GNN models on four out of five benchmarks leaving one program as a test set in a round-robin 'n-1'

training/test regime, and an extra unseen benchmark (control-sensitive inversek2j or data-sensitive LU) for validation to verify GLAIVE's transferablity. We compile each benchmark using `g++ 5.0` targeting `x86` architecture. For fault injection and ground truth generation we use gem5-Approxilyzer [12] in full-system mode using a Ubuntu-18.04 system image. All experiments were run on an Intel Xeon Gold 6242 CPU 16-core processor running at 2.8 GHz with 383 GB RAM.

ML algorithms: We compare GLAIVE with a set of other ML algorithms in terms of accuracy, speed, and transferability. Specifically, we use Multi-Layer Perceptron (MLP-BIT) classifier at bit-level, and Random Forest (RF-INST) and Support Vector Machine (SVM-INST) regressors at instruction-level. GLAIVE and MLP-BIT use the same bit-level node attributes and RF-INST uses instruction-level attributes of Table I. We use Sklearn with default parameters for baseline ML algorithms.

Metrics and learning parameters: We use standard accuracy score for bit-level classification whereas for instruction-level regression we use top-K coverage and program vulnerability loss to measure its performance. Accuracy can take value from 0 to 1 while values closer to 1 mean higher accuracy. For GLAIVE, we use 10 epochs, a batch size of 256, a learning rate of 0.001 with a hidden dimension of 128, cross entropy as loss function, and rectified linear unit (ReLU) as the activation function in a 3-layer GNN with neighbour sample size of 50.

V. Experimental results

A. Comparison of accuracy

In this experiment, we compare the fault estimation quality of GLAIVE to the MLP-BIT algorithm in terms of accuracy. We show accuracy for both GLAIVE and MLP-BIT on each of the benchmarks in Table III. GLAIVE achieves up to (0.684 - 0.525)/0.525 $\times$ 100% = 30.29% more accuracy (average (0.807 - 0.741)/0.741 $\times$ 100% = 8.91%) than MLP-BIT for data-sensitive benchmarks. The learned model achieves high accuracy on unseen programs indicating that the learned knowledge of bit-level vulnerability is transferable to unseen programs.

Additionally, GLAIVE and MLP-BIT achieve a comparable accuracy, up to 99% (average 96%) for control-sensitive benchmarks. We identify three key factors for both GLAIVE and MLP-BIT achieving high accuracy in control-sensitive benchmarks. *First*, control instructions have higher fault resiliency intrinsically. For example, consider `cmp R1, R2` where R1, R2 are 8-bit wide general-purpose registers and contain `8'b41` and `8'b20`, respectively. A single-bit upset at the i^{th} bit ($i \leq 5$) in `R1` will keep the output of faulty `cmp` same as that of fault-free `cmp` output. This renders many edges in bit-level CDFG useless for GLAIVE to learn neighborhood information, eventually reducing GLAIVE to MLP-BIT. *Second*, our fault model does not consider faults in the control registers. *Finally*, due to the limited number of control instructions as compared to a rich set of arithmetic/logic instructions, it is easier for both methods to learn different error propagation patterns.

These experiments show that GLAIVE can effectively learn the bit-level vulnerability and the learned knowledge on vulnerability is transferable to unseen programs.

Data-sensitive benchmarks			Control-sensitive benchmarks		
Name	GLAIVE	MBT	Name	GLAIVE	MBT
Blackscholes	0.92	0.91	Dijkstra	0.97	0.97
FFT	0.73	0.73	Streamcluster	0.98	0.99
Swaptions	0.89	0.87	Jmeint	0.95	0.95
Radix	0.86	0.75	Astar	0.99	0.99
Ctaes	0.68	0.53	Sobel	0.96	0.96
LU	0.77	0.68	Inversek2j	0.91	0.91

TABLE III: **Accuracy comparison of GLAIVE and MLP-BIT – MBT**: MLP-BIT.

B. Comparison of top-K coverage

In this experiment we compare effectiveness of GLAIVE and other ML algorithms in estimating top-K coverage for a given protection budget K. We vary K from 5%-100% in steps of 5% and measure top-K coverage for both bit-level and instruction-level ML algorithms on all benchmarks. For brevity, we show average top-K coverage of all control-sensitive benchmarks and top-K coverage of two representative data-sensitive benchmarks (Radix and Swaptions) in Figure 4.

In Figure 4a and 4b we observe that instruction-level algorithms achieve lower top-K coverage than bit-level algorithms, particularly with K value less than 70%. This implies that the instruction-level algorithms fail to learn bit-vulnerability and consequently failed to predict most vulnerable instructions. On the other hand, both GLAIVE and MLP-BIT learned fine-grain bit vulnerability and achieved higher average top-K coverage than instruction-level algorithms. This is consistent with the higher accuracy of GLAIVE as shown in Section V-A. GLAIVE achieves up to 4.89% (average 1.4%) and up to 8.03% (average 1.08%) higher top-K coverage in Swaptions and Radix respectively, than MLP-BIT. For control-sensitive benchmarks, GLAIVE and MLP-BIT achieve similar, on average 93% top-K coverage as shown in Figure 4c. GLAIVE achieves on average 90.23% top-K coverage for all benchmarks for varying protection budgets which is up to 21.3% and 23.18% higher than the top-K coverage of RF-INST and SVM-INST respectively.

This experiment shows that GLAIVE is effective in identifying vulnerable instructions compared to other ML algorithms.

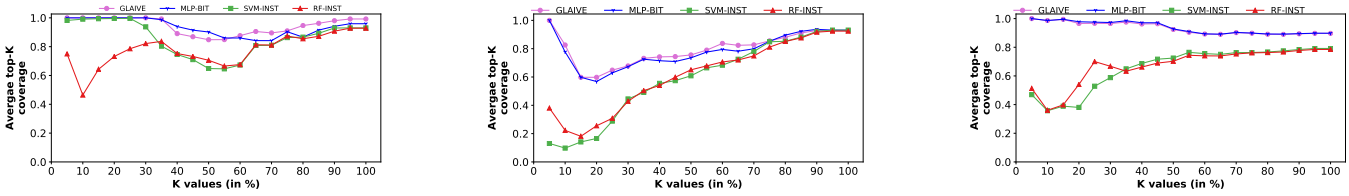
C. Comparison of program vulnerability error

In this experiment, we compare the program vulnerability error for each of the ML algorithms as compared to baseline FI. We observe from Figure 5a that for data-sensitive benchmarks, GLAIVE achieves on average 26.24%, 33.09%, and 16.78% lower error than RF-INST, SVM-INST, and MLP-BIT respectively. Additionally, for control-sensitive benchmarks, GLAIVE achieves on average 1% and 12.7% lower error than RF-INST and SVM-INST respectively, and less than 1% more error compared to MLP-BIT.

This experiment shows that GLAIVE is highly effective in accurately estimating program vulnerability compared to other ML algorithms.

D. Comparison of runtime

In this experiment, we compare the runtime of the instruction vulnerability estimation of GLAIVE to that of other ML algorithms considering FI as a baseline. Due to the large difference in the FI runtime and ML-based inference runtime, we show



(a) Radix (data-sensitive benchmark). (b) Swaptions (data-sensitive benchmark). (c) Control-sensitive benchmarks.

Fig. 4: **Comparison of top-K coverage for various ML algorithms for varying protection budget** K .

Data-sensitive benchmarks					Control-sensitive benchmarks				
Name	M1	M2	M3	M4	Name	M1	M2	M3	M4
D1	0.079	0.074	0.125	0.204	C1	0.053	0.052	0.213	0.070
D2	0.343	0.441	0.304	0.349	C2	0.033	0.001	0.127	0.036
D3	0.093	0.095	0.680	0.614	C3	0.098	0.093	0.258	0.083
D4	0.079	0.260	0.402	0.298	C4	0.019	0.015	0.239	0.100
D5	0.550	0.922	0.985	0.785	C5	0.082	0.081	0.173	0.058
D6	0.031	0.388	0.663	0.498	C6	0.181	0.180	0.217	0.157

(a) Program vulnerability error comparison

Data-sensitive benchmarks					Control-sensitive benchmarks				
Name	M1	M2	M3	M4	Name	M1	M2	M3	M4
D1	2.22	3.22	3.48	3.48	C1	2.3	2.5	2.8	2.8
D2	2.23	2.35	2.65	2.65	C2	2.27	2.45	2.75	2.75
D3	1.96	2.16	2.46	2.46	C3	2.45	2.79	3.09	3.09
D4	2.31	2.39	2.68	2.68	C4	2.46	2.67	2.97	2.97
D5	2.47	3.01	3.31	3.31	C5	2.48	2.71	3.01	3.01
D6	1.4	1.47	1.7	1.7	C6	2.38	2.69	2.99	2.99

(b) Instruction vulnerability estimation speedup comparison

Fig. 5: **Comparison of GLAIVE and each of the ML algorithms for different benchmarks in terms of key metrics – M1**: GLAIVE. **M2**: MLP-BIT. **M3**: SVM-INST. **M4**: RF-INST. **D1**: Blacksholes. **D2**: FFT. **D3**: Swaptions. **D4**: Radix. **D5**: Ctaes. **D6**: LU. **C1**: Dijkstra. **C2**: Streamcluster. **C3**: Jmeint. **C4**: Astar. **C5**: Sobel. **C6**: Inversek2j.

the speedup in the logarithmic scale in Figure 5b. We observe that ML based algorithms can achieve up to three orders of magnitude of a speedup as compared to baseline FI (even with 16-way parallel execution). Additionally, GLAIVE is slower than MLP-BIT as GLAIVE needs more computations due to its complex graph embedding structure. Since GLAIVE considers a larger bit-level dataset, it is up to one order of magnitude slower than RF-INST and SVM-INST. However, GLAIVE still achieves up to 221× speedup over accelerated FI.

This experiment shows that GLAIVE is computationally efficient for vulnerability estimation as compared to FI.

VI. Related Work and Conclusion

The literature for program resiliency can be classified into fault injection, analytical models, and ML-based approaches.

In [10] authors proposed an accelerated FI using a multi-level processor simulator. gem5-Approxilyzer [12] reduces FI complexity by identifying equivalent fault groups. However, FI methods are accurate but are extremely time-consuming.

To quickly estimate the effect of soft-errors on programs, various analytical models were proposed. CIAP [5] uses a combination of static analysis and runtime monitoring to identify critical instructions. Trident [8] constructs a three-level model to capture error propagation via static control, memory, and data dependency. However analytical models are inaccurate.

To leverage high accuracy of FI and high speed of analytical models, ML-based methods are increasingly used to identify vulnerable instructions. SVM models are used to predict critical instructions via partial FI [9]. Random Forest regressor is used to estimate the SDC probability per instructions [6]. However, there has not been an effort to automatically learn instruction vulnerability using graph-structured data.

Recently, GNN models have been applied to a plethora of difficult EDA problems. Ustun et al. [11] have developed a GNN model to automatically learn operation pattern mapping for high-level synthesis. In [1] authors proposed a GCN model to extract structural features from the gate-level netlist and predict soft error propagation metrics. Zhang et al. proposed a GPU-accelerated GNN model for fast, accurate, and transferable vector-based average power estimation [14].

In conclusion, we have presented GLAIVE, a GNN-based model to automatically learn instruction vulnerability patterns. Our empirical analysis showed that GLAIVE is accurate, scalable, fast, and transferable compared to the state-of-the-art FI, ML, and analytical methods. In future, we plan to improve GLAIVE to estimate instruction vulnerability in out-of-order processors, dynamic scheduling, and hybrid branch prediction.

References

[1] A. Balakrishnan, T. Lange, M. Glorieux, D. Alexandrescu, and M. Jenihhin. Composing graph theory and deep neural networks to evaluate seu type soft error effects. *MECO*, 2020.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, USA, 2011.

[3] H. Cai, V. W. Zheng, and K. C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[4] E. Cheng, Daniel-Mueller-Gritschneder, J. Abraham, P. Bose, A. Buyuktosunoglu, and et al. Cross-layer resilience: Challenges, insights, and the road ahead. *DAC*, 2019.

[5] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. *ICCAD*, 2011.

[6] J. Gu, W. Zheng, Y. Zhuang, and Q. Zhang. Vulnerability analysis of instructions for sdc-causing error detection. *IEEE Access*, 2019.

[7] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *NIPS*, 2017.

[8] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. Modeling soft-error propagation in programs. *DSN*, 2018.

[9] L. Liu, L. Ci, W. Liu, and H. Yang. Identifying sdc-causing instructions based on random forests algorithm. *TIIS*, 2019.

[10] D. Mueller-Gritschneder, U. Sharif, and U. Schlichtmann. Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator etiss-ml. *ICCAD*, 2018.

[11] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang. Accurate operation delay prediction for FPGA HLS using graph neural networks. *ICCAD*, 2020.

[12] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve. gem5-approxilyzer: An open-source tool for application-level soft error analysis. *DSN*, 2019.

[13] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 2017.

[14] Y. Zhang, H. Ren, and B. Khailany. Rannite: Graph neural network inference for transferable power estimation. *DAC*, 2020.

[15] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *CoRR*, 2018.