Journal of Electronic Testing (2022) 38:303–319
https://doi.org/10.1007/s10836-022-06005-y

Deep Soft Error Propagation Modeling Using Graph Attention Network

Junchi Ma[1] · Zongtao Duan[1] · Lei Tang[1]

Received: 2 March 2022 / Accepted: 27 May 2022 / Published online: 8 June 2022

Abstract
Soft errors are increasing in computer systems due to shrinking feature sizes. Soft errors can induce incorrect outputs, also called silent data corruption (SDC), which raises no warnings in the system and hence is difficult to detect. To prevent SDC effectively, protection techniques require a fine-grained profiling of SDC-prone instructions, which is often obtained by applying machine learning models. However, these models rely on handcrafted features, and lack the ability to reason about SDC propagation, which leads to an inferior SDC prediction performance. We propose a novel *G*raph *A*ttention ne*T*work to *P*redict *S*DC-prone instructions (GATPS). The GATPS representation is a heterogeneous graph with different types of edges to represent various instruction relations. By stacking layers in which nodes are able to attend over their neighborhoods' features, GATPS automatically captures the structural features that contribute to SDC propagation. The attention mechanism is applied to compute the importance values to the neighboring nodes, which quantifies the fault effect on the neighboring nodes. Moreover, the inductive model of GATPS can be applied to unseen programs without retraining, and it requires no fault injection information of the target program. Experiments revealed GATPS achieved a 34% higher F1 score compared to the baseline method and a 40-fold speedup compared to the fault injection approach.

Keywords Silent Data Corruption · Soft Error · Graph Neural Network · Attention Mechanism · Inductive Learning

1 Introduction

Soft error is one of the major issues future computing systems face [8]. Researchers have witnessed unacceptably high failure rates when running scientific workloads on ground-level cloud or grid systems [5]. One outcome of soft errors is silent data corruption (SDC), which means corrupted output is produced without any trace of failure, representing the worst case scenario for a resiliency solution [15]. Due to no trace of failure, SDC is difficult to detect [18]. When corrupted output are used to make decisions in areas such as climate [2], it can lead to catastrophic consequences. Facebook observes that CPU SDCs in global data centers are orders of magnitude higher than soft-error based simulations [3]. With increased silicon density and technology scaling, methods should be invested to counter these issues.

Researchers have demonstrated that a small proportion of the instructions contribute to a majority of SDCs [26], which facilitates selective instruction duplication techniques to protect only partial instructions to maximize the SDC coverage with low performance overhead [13, 17]. A precise profiling of the SDC-prone instructions is required before deciding which instructions should be protected. This is the goal of our work.

The straightforward way to achieve the goal of profiling the SDC-prone instructions is using fault injections [10], which is often implemented by software to emulate faults at assembly level. Real-world programs may consist of millions of dynamic instructions, which makes fault injections very time-consuming. Performing pure fault injection to determining whether each instruction of a program is SDC-prone is too expensive to be practical.

There has been a series of work focused on predicting SDCs to minimize the injection overhead [14, 19, 27]. Fault injections are applied to a portion of instructions in a program to create a training set to fit the parameters of the machine learning model. The model is then used to conduct predictions on the other instructions. The input of machine learning model is the structural features which describe the

Responsible Editor: A. Yan

Junchi Ma
majunchi@chd.edu.cn

1 School of Information Engineering, Chang'an University, Middle Section of Nan'erhuan Road, Xi'an, Shaanxi, China

fault propagation context of an instruction. The structural features are important to build an effective machine learning model. Usually the structural features are instruction statistics extracted by analyzing the assembly file [14].

Unfortunately, such approaches have two inevitable shortcomings. First, researchers have to decide what information summarizes the context of an instruction in a way that may be useful in describing the fault propagation process. The occurrence of SDC has been revealed to be affected by complicated semantics including static data dependency [16], control flow [20], memory layout [15], etc. These prior approaches are not capable of describing the complicated semantics, and thus leading to inaccurate analytical models for fault propagation. Second, these approaches lack the ability of generalization, i.e., the trained model cannot be used to predict unseen programs, so the model has to be trained individually by using the fault injection data for each program. The major time cost is related to the size of the training set, determining how many fault injections are conducted. For example, the percentage of the instructions required to generate a training set for proPV model is 60% [27], thus the total time cost is at least 60% of the time cost of full fault injections, which may still be too high for large-scale programs.

To address the first shortcoming, graph neural network (GNN) can remove human involvement from SDC prediction task. In this paper, we apply the *G*raph *A*ttention ne*T*work to *P*redict *S*DC-prone instructions (GATPS) model. We see the program as a graph, and see the instruction as a node. Low-dimensional embeddings of nodes are learned during training and used for predicting SDC proneness of the corresponding instruction. Therefore, the task of SDC-prone instruction prediction is transferred to node classification in a graph neural network. Compared to prior work, the significant difference is that GATPS does not define explicit structural features. GATPS learns the hidden structural features by aggregation of the neighboring nodes' features. To reason about fault propagation, we utilize an attention strategy to quantify the fault effect on the instructions. The self-attention strategy can evaluate the contributions of neighboring nodes to the fault propagation. To model the different effects of the data flow and control flow, we apply heterogeneous graphs to represent the various types of relations between the instructions and train separate attention coefficients for different types of edges.

To address the second shortcoming, we then extend our GATPS model to the task of inductive learning to make predictions for an unseen program, i.e., the testing program remains completely unobserved during training. The learned knowledge of the soft error propagation is transferable to other programs, and thus, the trained model can be used to predict SDC-prone instructions of other programs. Since the training set is not related to the target program, the inductive model does not require any fault injections on the target program. By applying the inductive model, we can decouple the vulnerability assessment from fault injections, which facilitates quick and accurate instruction vulnerability estimation.

In summary, the main contributions of this paper are as follows:

1. We propose GATPS, which applies a graph attention network framework to obtain an accurate profiling of SDC-prone instructions. The model automatically captures structural features of instructions by aggregating neighboring information. Our approach achieves an end-to-end manner of learning, minimizing the influence of researchers' understandings of fault propagation.
2. The inductive learning of GATPS is designed to conduct predictions without fault injections on target programs, thus enables decoupling of the vulnerability assessment from the fault injections. This technique eliminates the requirement of fault injections on target programs for vulnerability assessment.
3. The results of transductive learning, inductive learning, and the homogeneous GATPS graph model are compared, and the advantages and suitable application scenarios are discussed. We also propose a customized SDC prediction strategy based on provided data and user requests. In particular, when lacking fault injection information of the target program, we can still obtain a relatively high prediction accuracy.

2 Related Work

In order to predict the SDC vulnerability of instructions, prior work has employed classification and regression tree (CART) [27], deep forest regression [19], and support vector machine (SVM) [14]). The structural features, used as the input of machine learning algorithms, were defined by researchers to depict the context of fault propagation. We enumerate the structural features defined by prior work in Table 1. These structural features can be categorized into basic block(bbl, denotes a single entrance, single exit sequence of instructions), function, and data-chain based on the granularity of code.

IPAS defined the structural features in terms of the number of instructions (#inst) in a scope of bbl or local function [14]. #inst reflected the distance between the fault activation site and the end of the bbl or function. However, many intermediate instructions included in #inst were probably irrelevant to the fault propagation. For example, *nop* instruction does nothing during execution and it does not participate in the fault propagation. To address this issue, features describing the data dependence were introduced. The fault propagation paths could be dissected in the perspective of

Table 1 Structural features defined by related work

Structural Features			proPV	DFRMR	IPAS	SDCTune	PARIS
Basic block level	#inst of the bbl		√	√	√		
	#remaining inst		√	√	√		
	#cmp inst		√				
	#branch inst		√		√		
Function level	#inst of the function				√		
	#remaining inst to ret inst				√		
	#bbl of the function				√		
	#successor bbl		√	√	√		
	#predecessor bbl			√			
		connector inst	√				
Data flow level	whether the inst data feeds	cmp inst				√	√
		store inst				√	√
		shift inst					√
		add inst					
		address-related inst		√			

data flow which revealed the data-dependent instructions. DFRMR proposed new structural features whether the result of instruction was supplied to any address-related instructions [19]. An erroneous address was probably to result in a crash, so introducing the address-related features helped to filter out crash-prone instructions. proPV defined the connector instructions [27], which operated returned values of invoked functions, function parameters, or global variables. These variables represented the data transfer between functions, so the correctness of connector instructions reflected the fault propagation between functions. Whether the fault activated in one function propagated to other functions was determined by checking the status of related connector instruction. PARIS constructed features based on six resilience computation patterns (such as dead locations, repeated addition, conditional statements, and data truncation) related to application resilience [7]. These patterns may not be able to cover all propagation situations yet. SDCTune constructed a classification tree to categorize the stored values by identifying whether they were used for addressing or comparison operations [20]. Linear regression was applied upon certain leaf nodes of the classification tree which contained continuous features such as data width to compute the probability of it causing SDC.

These works selected various structural features to dissect the propagation context. To date, these structural features were chosen based on researchers' understanding of fault propagation. However, high accuracy loss illustrated that the handcrafted features may not cover the valuable information for predicting SDC. The proposed structural features depicted certain aspects of the data flow or control flow of a program, but they were not adequate to depict the propagation of SDC. Many factors that were validated to have effects on SDC (such as application-specific behaviors [9] or stack behaviors [22]) were not included in aforementioned features. It can be inferred that there probably exist more obscure factors that contribute to SDC. Enumerating exhaustively valuable structural features tends to be impractical. How to depict the execution context of instruction for predicting SDC proneness still remains unsolved.

GNN has been successfully applied to tackle problems such as node classification and community discovery. Message passing methods were widely adopted by GNN to learn node embeddings by aggregating the information from the target node's neighbors. GRAPHSAGE learned the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood [11]. GRAPHSAGE applied a message passing method of uniformly sampling one node's neighbors, and in many graph applications each neighbor may contribute differently. To address this shortcoming of GRAPHSAGE, graph attention network (GAT) explored attention mechanism, enabling assigning different importance values to different neighbors [25]. To perform graph embedding in heterogenous graphs, heterogeneous graph structural attention neural network (HetSANN) proposed a type-aware attention layer which jointed different types of neighboring nodes [12]. Relational Graph Convolutional Networks (R-GCN) used multiple weight matrices to project the node embeddings into different relation spaces to capture the edge heterogeneity [24].

We find GNN is suitable to solve above problems. Message passing methods for GNN can be used to extract the complex context of instruction. Moreover, attention mechanism is appropriate to learn the fault propagation probabilities automatically. The embedding for each node can reflect both the attribute feature of the corresponding instruction and its execution context.

3 Model

We use the code example in Fig. 1 to explain a fault propagation process and present our model. A simple snippet of code is shown in subgraph (a), which computes the summation of array A. The assembly code is shown in subgraph (b). An error in one instruction either propagates to downstream instructions or stops to propagate. We take the propagations between instructions 5 → 6, 2 → 3, 7 → 8 as an example. Instruction 6 directly loads the execution result of instruction 5 (*[ebp-0* × *8]*). When a soft error activates in instruction 5, the error propagates to instruction 6. For the propagation 2 → 3, since the result of instruction 2 is used for addressing by instruction 3, it is likely that the corrupted address causes a crash and the fault propagation is ended. For the propagation 7 → 8, instruction 7 conducts a comparison operation and affects the consequent branch instruction *jl*. The error in instruction 7 may not change the comparison result so the error is masked which ends the fault propagation. For example, *[ebp* + *0xc]* = 10, *eax* = 2, the least significant bit of *eax* is flipped from 0 to 1, changing its value to 3. However, *eax* < *[ebp* + *0xc]*, so it still selects L1 branch. To conclude, due to the different relations, the effects of fault propagations 5 → 6, 2 → 3, 7 → 8 vary. So a heterogeneous graph network is a natural way of representing multi-relational fault propagations. We build a graph shown in subfigure (c). Each node in the graph represents a dynamic instruction and the edge represents the relation between instructions. Different types of edges are used to represent the different relations between (5,6), (2,3) and (7,8), which is denoted by different colors. We also use edge weight to denote the fault effects, which can be learned automatically by attention mechanism.

In this paper, we intend to learn the low-dimensional embeddings for each node and apply it for downstream

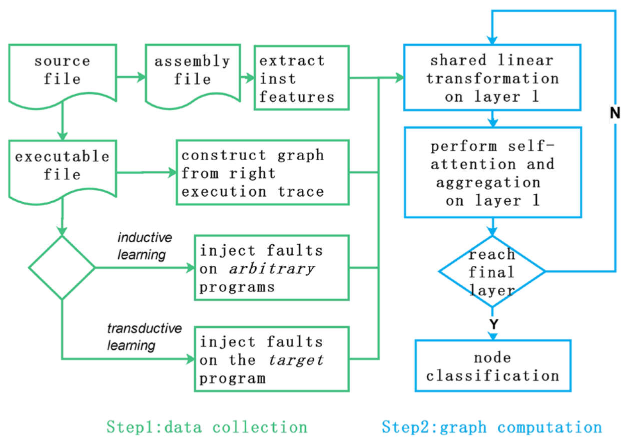
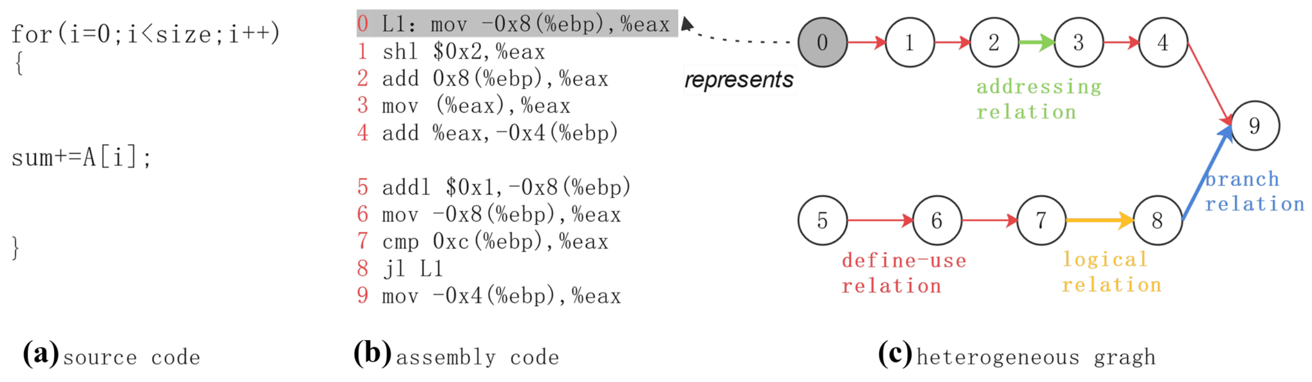

source file
assembly file
extract inst features
executable file
construct graph from right execution trace
inductive learning
inject faults on *arbitrary* programs
transductive learning
inject faults on the *target* program
shared linear transformation on layer l
perform self-attention and aggregation on layer l
reach final layer
N
Y
node classification
Step1:data collection
Step2:graph computation

Fig. 2 Overall framework of GATPS (*G*raph *A*ttention ne*T*work to *P*redict *S*DC-causing instructions) model

node classification tasks. Although there are various GNN models, they can be broadly grouped into transductive learning and inductive learning models. A transductive model must 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 general knowledge via 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. Specifically, transductive learning requires SDC labelling of instructions of a target program for training, so fault injections need to be conducted on the target program. Inductive learning does not require fault injections on the target program. The inductive model can extract a generalized mechanism of SDC propagation and apply it to an unknown program.

We construct both a transductive model and an inductive model to predict SDC-prone instructions. Figure 2 shows the overall framework of our proposed GATPS model. The input



```
for(i=0;i<size;i++)
{

   sum+=A[i];

}
```

```
0 L1: mov  -0x8(%ebp),%eax
1 shl  $0x2,%eax
2 add  0x8(%ebp),%eax
3 mov  (%eax),%eax
4 add  %eax,-0x4(%ebp)

5 addl $0x1,-0x8(%ebp)
6 mov  -0x8(%ebp),%eax
7 cmp  0xc(%ebp),%eax
8 jl  L1
9 mov  -0x4(%ebp),%eax
```

Fig. 1 Example of fault propagation and proposed graph representation

of GATPS is the source code of program and SDC labels, and the output is the SDC prediction of instructions. The processing of GATPS can be divided into data collection and graph computation. As we apply both the transductive and inductive models, the data collection phases of these two approaches are different. For the transductive model, we apply partial fault injections on the target program to obtain the training dataset (experiments showed that 30% was sufficient to obtain a stable prediction accuracy), and for the inductive model, we inject arbitrary programs, which do not include the target program. Our inductive model is able to learn the general propagation knowledge and apply it on the target program. The graph computation is processed through L attention layers, which are described in detail in Sect. 3.2. The output of the previous attention layer is fed to the next attention layer. The output of the final attention layers is then supplied to the softmax function to predict the node label.

In this section, we present the attention layer used to construct graph attention networks and the final layer for node classification.

3.1 Graph Construction

A graph comprises a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$ between nodes, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node represents an executed instruction and also defines a computational graph based on the neighboring nodes. Edges are typed to differentiate instruction relations and we denote the edge types by $\mathcal{A}$. An edge e is mapped to a certain type $p \epsilon \mathcal{A}$ by using $\phi(e) = p$, where ϕ is the mapping function from $\mathcal{E}$ to $\mathcal{A}$. We consider following four types of edges.

1. Branch relation. The relation depicts the impact of *branch* instructions on the its consequently executed instructions. If e_{ij} satisfies branch relation, instruction i has to be a *branch* instruction, and instruction j is i's consequent instruction. When the *branch* instruction i is affected by soft error, an incorrect branch may be taken during execution and affects instruction j.
2. Addressing relation. If the destination operand of instruction i is used for addressing by instruction j, e_{ij} satisfies addressing relation, $2 \rightarrow 3$ in Fig. 1 is an example of addressing relation.
3. Logical relation. The conditional jump instruction decides whether to take the branch by checking the status register *eflags*, and the value of *eflags* is determined by the comparison result of *cmp* instruction. We have given an example of logical relation $7 \rightarrow 8$ in Fig. 1 and show that flipping the operand value does not affect the value of *eflags*. The logical relation may lead to masking the error of *cmp* instructions.
4. Define-use relation. If instruction j reads the data that instruction i writes, then e_{ij} satisfies define-use relation. To distinguish from other relations, the data operated by instruction j is neither *eflags* nor used for addressing.

These edge types cover major instruction relations, which are used in analyzing SDC propagation [15, 16, 20]. We show these types of edges by using different colors in Fig. 1. The branch relation describes the control flow propagation, and the other three relations describe data flow propagation. Faults in data used for addressing relations cause crashes easily since they likely generate invalid addresses. Faults in data used for logical relations may not cause different values of *eflags* and thus can easily cause benign outcomes. Therefore, these relations vary significantly in the patterns of propagating to incur SDC, which facilitates reasoning about SDC propagation. The attention strategies are designed for the aggregation of neighboring nodes with different edge types separately.

We construct the graph by using the traces of dynamic instruction in the execution, and detailed information is shown in Algorithm 1. The dynamic instructions' execution information can be recorded by using injection tool. Each time we take one trace item, and generate node and edges corresponding to the trace item. The types of edges are determined by checking which definition of edge relation

Algorithm 1: Graph Construction Algorithm

```
Input: Dynamic instructions' trace
Output: $\mathcal{G}$: a heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
1.  foreach instruction item i in trace:
2.    $\mathcal{V} = \mathcal{V} \cup v_i$ //generate a new node $v_i$
3.    if i is branch instruction then
4.      foreach instruction j in next basic block:
5.        $\mathcal{E} = \mathcal{E} \cup e_{ij}$ // generate a new edge $e_{ij}$
6.        $\phi(e_{ij})$=branch relation
7.    if i is conditional jump instruction then
8.        // get the last inst writes EFLAGS
9.        k=last_write(EFLAGS)
10.       $\mathcal{E} = \mathcal{E} \cup e_{ki}$ //generate a new edge $e_{kj}$
11.       $\phi(e_{ki})$=logical relation
12.   if i contains base or scale register b then
13.       //get the last inst writes b
14.       m=last_write(b)
15.       $\mathcal{E} = \mathcal{E} \cup e_{mi}$ //generate a new edge $e_{mi}$
16.       $\phi(e_{mi})$=addressing relation
17.   else foreach source operands of instruction i :
18.       //get the last inst writes source operands
19.       w=last_write(source operand of i)
20.       $\mathcal{E} = \mathcal{E} \cup e_{wi}$ //generate a new edge $e_{wi}$
21.       $\phi(e_{wi})$= define-use relation
22.   update last_write(i's destination operand)=i
```

matches the dynamic instruction's operation. If the instruction is branch instruction, we create an edge between the branch instruction and each instruction in the consequent basic block (line 3–6 in Algorithm 1). Then we use last_write(r) to record the instruction that last writes the *r* (register or memory location), thus we are able to create connection between the last instruction that writes *r* and the instruction that reads *r*. For example, if *r* is base register (used for addressing), the edge is categorized as addressing relation. Thus new edges are generated and categorized into addressing relation, logical relation and define-use relation (line 7–21 in Algorithm 1).

3.2 Attention Layer

In this subsection, we describe the structure of a single attention layer. The number of nodes of single layer equals to the number of dynamic instructions. Multi-head attention mechanism is employed by each attention layer for the stabilization of the learning process of self-attention.

The input to the first layer is a set of node features, $h = \{h_0^{(0)}, h_1^{(0)}, \dots, h_N^{(0)}\}$, $h_j^{(0)} \in \mathbb{R}^F$, where *N* is the number of nodes, and *F* is the number of attribute features in each node. The attribute features contain two indices, representing the opcode and the destination operands of the corresponding instruction. The opcode denotes how the instruction affects data or execution of other instructions, and the destination operands denote which data the instruction affects. The destination operands we consider include the registers *eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags*, and *eip*. If the instruction uses an 8-bit register, such as *ah*, or the 16-bit register *ax*, we translate it to 32-bit *eax*. The registers are encoded as unique one-hot vectors. The opcode and destination operands of each instruction are converted to a vector by searching the embedding table.

The workflow of an attention layer in the GATPS is shown in Fig. 3. We consider a node $j \epsilon \mathcal{V}$ represented as $h_j^{(l)}$ in the *l*-th layer. $h_j^{(l+1,m)}$ denotes node *j*'s hidden state outputted by the attention head *m* of the (*l* + *1*)-th attention layer. Each attention layer performs a transformation operation and an aggregation operation of neighborhood. To accumulate from the node's previous layer representation, we add an identity matrix to the adjacency matrix, thus adding virtual self-loops to the graph. We use $\mathcal{N}_j$ to denote the neighbors of node *j* in the graph, after adding self-loops $\mathcal{N}_j = \{i, j, q, r\}$.

As an initial step, a shared linear transformation parameterized by a weight matrix, $W_{\phi(e_{ij})}^{(l+1,m)}$, is applied to each neighboring node $i \in \mathcal{N}_j$ as follows:

$$h_{i,e_{ij}}^{(l+1,m)} = W_{\phi(e_{ij})}^{(l+1,m)} h_i^{(l)}, \tag{1}$$

where $h_{i,e_{ij}}^{(l+1,m)}$ is the projection from layer *l*'s embedding to the space of node *i* in the (*l* + *1*)-th attention layer of the *m*-th head. For the first layer, the input embedding represents the attribute features, and the weight matrix $W_{\phi(e_{ij})}^{(2,m)} \in \mathbb{R}^{F' \times F}$, where F' is the number of features in each node after linear transformation. For the other layers, $W_{\phi(e_{ij})}^{(l+1,m)} \in \mathbb{R}^{F' \times F'}$. Note that the weight matrix $W_{\phi(e_{ij})}^{(l+1,m)}$ also consider the type of edge since the equation contains $\phi(e_{ij})$. As shown in Fig. 3, $\phi(e_{ij}) = \phi(e_{rj}) =$ define-use relation, e_{ij} and e_{rj} share the same weight matrix $W_{\phi(e_{ij})}^{(l+1,m)}$.

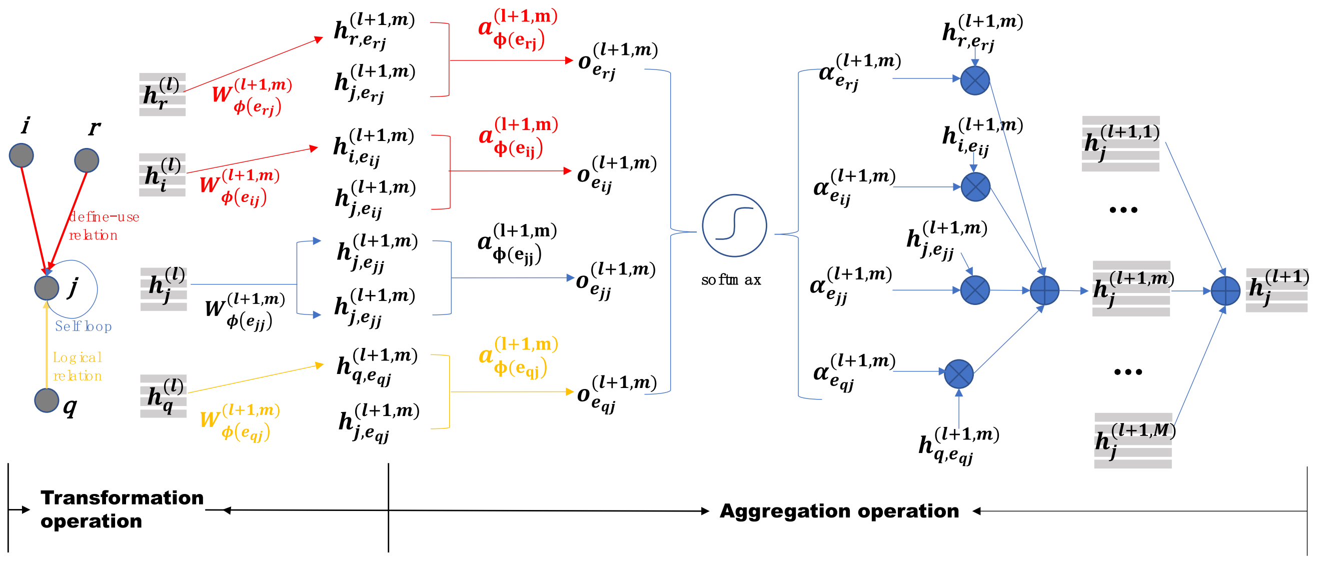


Fig. 3 Workflow of an attention layer in GATPS

We then perform self-attention mechanism to learn a representation for each node. The self-attention mechanism computes the interaction between input node features. Applying such mechanism makes it possible to concentrate more on important node features. The self-attention mechanism yields aggregates of the interactions and attention coefficients. The attention mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ is a dot-product function and shared by the edges of the same type. In Fig. 3, e_{ij} and e_{rj} share the same attention parameter $a^{(l+1,m)}_{\phi(e_{ij})} = a^{(l+1,m)}_{\phi(e_{rj})}$.

$$o^{(l+1,m)}_{e_{ij}} = \sigma\left(a^{(l+1,m)}_{\phi(e_{ij})}(h^{(l+1,m)}_{i,e_{ij}}, h^{(l+1,m)}_{j,e_{ij}})\right), \tag{2}$$

where σ is a LeakyReLU(.) activation function. The attention coefficient $o^{(l+1,m)}_{e_{ij}}$ indicates the importance of node *i* to the target *j*. For the downstream prediction task, the importance denotes the fault effect of instruction *i* on the instruction *j* through the fault propagation path e_{ij}. The learning of importance quantifies the fault effects on the neighboring nodes. Moreover, since the attention coefficients are trained for each type of edge, the model is able to characterize various patterns of fault propagations. We perform masked attention to apply the real graph structure, which only computes the node embedding if two nodes are connected. To make importance values comparable across different nodes, normalization is performed on these importance values across all edges to node *j* by using a softmax function:

$$\alpha^{(l+1,m)}_{e_{ij}} = e^{o^{(l+1,m)}_{e_{ij}}} / \sum_{k \in N_j} e^{o^{(l+1,m)}_{e_{kj}}}. \tag{3}$$

With the weights of each edge associated with node *j*, the aggregation for node *j* can be performed as

$$h^{(l+1,m)}_{j} = \sigma\left(\sum_{k \in N_j} \alpha^{(l+1,m)}_{e_{kj}} h^{(l+1,m)}_{j,e_{kj}}\right). \tag{4}$$

After computing the representations under M attention heads, the representations of node *j* are concatenated, and then outputted by the $(l + 1)$-th attention layer:

$$h^{(l+1)}_{j} = \|_{m=1}^{M} h^{(l+1,m)}_{j}, \tag{5}$$

where $h^{(l+1)}_{j} \in \mathbb{R}^{MF'}$ and ‖ denotes the concatenation operation. The low-dimensional representations outputted by the final layer is then used for prediction. For the final layer, we average the representations under M attention heads as follows:

$$h^{(l+1)}_{j} = \frac{1}{M} \sum_{m=1}^{M} h^{(l+1,m)}_{j}. \tag{6}$$

For SDC prediction task, a softmax function is applied in the end to produce the outcome labels. The softmax function outputs probability of incurring each outcome class (SDC, benign, crash and hang). Using labeled data, we minimize the cross-entropy loss:

$$\mathcal{L} = -\sum_{j \in \mathcal{V}_l} y_j \, log \, \tilde{y}_j, \tag{7}$$

where y_j is the outcome class label gained from fault injection, and $\tilde{y}_j$ is the predicted class label for node *j*. $\mathcal{V}_l$ denotes the set of labeled nodes.

4 Experimental Setup

4.1 Fault Model & Injection Infrastructure

We considered a single bit flip that occurred in the register file or memory. The dynamic instrument framework Pin was used to build fault injector tools [21]. Pin is a dynamic binary instrumentation framework for the IA-32 and × 86–64 instruction-set architectures, facilitating the creation of fault injection tools. During each run of fault injection, we altered one bit of the destination operand of the target instruction. A total of over 760,000 fault injections were performed in the experiment. The experiment was conducted on a rack server with an Intel(R) Xeon(R) CPU E5-2690 v3 processor operating at 2.60 GHz with 256 GB of memory.

The definitions of injection outcomes are shown in Table 2. The error code was recorded after execution to determine the causes of the crash. A successful execution returns 0, while an unsuccessful execution returns a non-zero value. For example, error code 139 denotes segmentation error.

The classes of outcomes were represented by one-hot encoding such that the shape of the tensor of classes was 1 × 4. For example, if the outcome was SDC, the outcome was encoded as [1,0,0,0]. A dataset with labels which denote the outcome was generated after processing the data of fault injections.

4.2 Application Program

We chose six benchmarks from Siemens benchmark suite. The benchmarks included *replace* (which performed string matching and replacement), *schedule* and *schedule2* (which performed management of scheduling), *print_tokens* and

Table 2 The definition of injection outcomes

Category	Definition	Criteria
Benign	the program produces the correct output	EC = 0 & Output = OFF
SDC	the program produces an erroneous output	EC = 0 & Output ≠ OFF
Crash	the program stops execution	EC ≠ 0
Hang	the program cannot finish execution	ET > threshold

EC Error code, *OFF* Output in fault-free execution, *ET* execution time

Table 3 Statistics of programs studied in our experiments

Program	#nodes	#edges	#faults injected
replace	6958	18191	33030
schedule	6809	16380	43660
print_tokens	2904	6995	18490
print_tokens2	5067	11394	33760
tot_info	5045	11486	18930
kmp	1360	2527	7520
dfs	12001	23508	66360
schedule2	6651	15664	39250
(under 11 separate inputs)	9166	17317	51740
	10003	18857	56330
	8703	16197	46880
	9897	18592	53920
	10387	19907	60640
	9462	17462	50010
	9356	17360	49750
	8330	15419	44050
	5280	10203	32450
	10213	19538	59370

print_tokens2 (which performed lexical analysis), and *tot_info* (which computed statistics for the input data). The classical algorithm *dfs* (which performed a deep first search on the map) and *kmp* (string-searching algorithm) were used to test the model. These programs contained several hundred lines of code with large test suite, and applied the C output function *printf* to print the output. The statistics of programs are shown in Table 3. We performed fault injections on 11 separate inputs of schedule2 to determine if our inductive model could learn from the results of the different inputs. The SDC rates were statistically significant with an error bar ranging from 0.94% (*kmp*) to 0.31% (*dfs*) at the 95% confidence intervals.

4.3 Parameters Settings for GATPS

4.3.1 Transductive learning

We adopted a GATPS model with L = 2, where L represented the number of attention layers. The first layer employed 8 attention heads and outputted F' = 8 features. ELU nonlinearity was applied as the activation function. The second layer was used to predict the outcome with a single attention head. All model was implemented in TensorFlow (version 1.14.0) with the Adam optimizer [1]. The learning rate was set to 0.005. The attention coefficients are initialized by Glorot initialization [6].

We split the labeled dataset obtained from the fault injections evenly split into a training set, validation set, and test set. We selected the one with best performance in the validation set and then evaluated them on the test set. For all models, the average performance of five repeated processes was calculated. We ran 100 epochs for the models.

4.3.2 Inductive Learning

For the inductive learning task, we applied a three-layer GATPS model. The first two layers consisted of M = 4 attention heads computing F' = 8 features, followed by an ELU nonlinearity. The final layer applied single attention head followed by a softmax activation function for classification. The attention coefficients are initialized by Glorot initialization [6]. The training sets for this task were sufficiently large, and we found no need to apply L2 regularization or dropout. We utilized a batch size of one graph during training. The inductive models were trained for 100 epochs.

4.4 Comparison Baseline

We compared to IPAS and proPV, two state-of-the-art machine learning models to predict SDC-prone instructions. IPAS applied SVM with a radial basis function kernel [14] and proPV applied a CART algorithm with the gini criterion [27]. The input features for the baseline machine learning algorithms comprised the structural features listed in Table 1 and the attribute features of the instruction. The attribute features mainly contained the instruction type and the size of the instruction's return value. The training ratio was set to the same configuration of GATPS model.

To evaluate the benefits of applying heterogeneous graphs in the model, we provided the results when a homogeneous graph (GATPS-homo) was applied, i.e., no types of edges were considered. Moreover, to evaluate the benefits of applying an attention mechanism in this setting, we also provided

Table 4 Performance indices of SDC prediction

Indices	Description
TP	True positive: # of pairs correctly classified as SDC-prone inst
FP	False positive: # of pairs incorrectly classified as SDC-prone inst
FN	False negative: # of pairs incorrectly classified as non-SDC-prone inst
precision	precision = TP/(TP + FP)
recall	recall = TP/(TP + FN)
F1	F1 = 2*precision*recall/(precision + recall)

Table 5 Comparison of results for SDC prediction in datasets

Dataset	*print_tokens*			*replace*			*schedule*			*schedule2*			*print_tokens2*			*tot_info*		
Metrics	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
GATPS	**0.89**	**0.98**	**0.93**	**0.84**	**0.90**	**0.87**	**0.88**	**0.81**	**0.84**	**0.87**	**0.90**	**0.88**	**0.83**	**0.79**	**0.81**	**0.94**	0.95	**0.94**
GATPS-homo	0.71	**0.98**	0.82	0.72	0.72	0.72	0.70	0.69	0.70	0.84	0.77	0.81	0.74	0.68	0.70	0.83	**0.96**	0.89
GATPS-const	0.49	0.88	0.63	0.57	0.63	0.60	0.68	0.55	0.61	0.75	0.71	0.73	0.58	0.58	0.58	0.82	**0.96**	0.88
IPAS	0.50	0.93	0.65	0.28	0.25	0.27	0.24	0.38	0.33	0.23	0.62	0.34	0.23	0.37	0.29	0.66	0.82	0.73
proPV	0.49	0.54	0.51	0.62	0.40	0.49	0.44	0.42	0.43	0.58	0.57	0.58	0.46	0.59	0.52	0.64	0.83	0.72

The bold values are the highest value among all results

the results when a constant attention mechanism (GATPS-const) was used with the same architecture–this assigned the same weight ($a^{(l+1,m)}_{\phi(e_{ij})} = 1$) to every neighbor.

5 Results

We adopted the precision, recall, and F1 score, which were commonly used metrics, to measure the prediction results. The false positive cases caused increased detection costs (duplication instructions), and the false negative cases caused a decrease in the SDC coverage. The F1 score considers both the SDC coverage and detection cost, ensuring a fair comparison between these techniques. The definitions are shown in Table 4.

5.1 Results for Transductive Learning

5.1.1 Prediction Performance

Table 5 shows a comparison of the results of our models with other baselines. Our model achieved the highest precision, recall, and F1 score for all studied programs. The average F1 score of the GATPS was 0.88. The average precision of our model was 52% higher than that of IPAS and 34% higher than that of proPV. Moreover, the average F1 score of our model was 44% higher than that of IPAS and 34% higher than that of proPV, showing that employing a framework of graph neural network can be beneficial.

The comparison of GATPS, GATPS-const, and GATPS-homo showed that utilizing the attention mechanism and heterogeneous graph improved the prediction accuracy. The GATPS model improved the F1 score by 20% compared to the GATPS-const model (the identical architecture with constant attention mechanism), directly demonstrating the significance of being able to assign different weights to different neighbors. Moreover, the average F1 score of the GATPS model was 12% higher than the GATPS-homo model with the input of the homogeneous graph, which indicated that the heterogeneous graph yielded a better representation of the various types of fault propagation. Various types of edges in the heterogeneous graph were naturally modeled in distinct spaces, which preserved the semantics of the different relations between the nodes.

The learned node embeddings can be visualized by t-SNE tool [23], which is straight-forward to find out how successful is the prediction results. The inferred node embeddings were embedded into a two-dimensional space. The SDC prediction results of the program *schedule2* are presented in Fig. 4. Color of node represented the outcome class. Red nodes represented the real SDC-prone instructions obtained in the fault injections. The points which were close to one another in the high-dimensional prediction results tended to be close to one another in the t-SNE plot. Figure 4 showed discernible clusters in the projected 2D space, that points from the same outcome class (e.g. red nodes) tended to be grouped close to one another. Figure 4 demonstrates GATPS's ability to learn meaningful embeddings for the fault propagations and make good predictions on SDC.

Moreover, we visualize the attention coefficients to show how GATPS discriminates between fault propagations. Attention coefficients are the key parameters that quantify

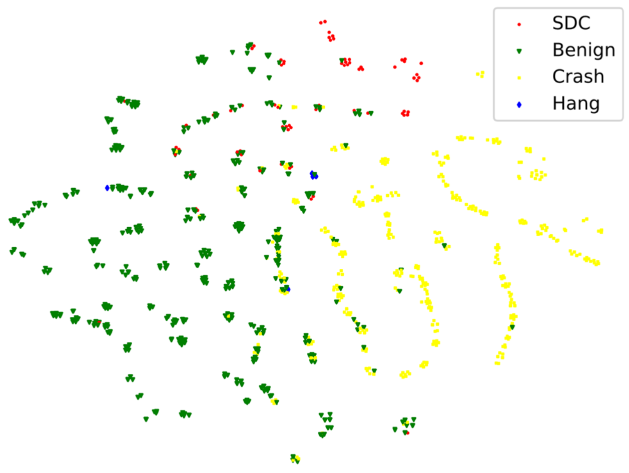


Fig. 4 t-SNE plot of the computed feature representations of the GATPS's final layer on the *schedule2* program. Node colors denote outcome classes

the fault propagations from one node to another (described in Sect. 3.2). The variations of the attention coefficients of attention head 0 for *schedule2* are shown in Fig. 5. The edge thickness indicates the aggregated normalized attention coefficients between the two nodes. A thicker edge means a larger attention coefficient, i.e. a stronger influence on the neighboring node. The thicknesses of the edges varied significantly, which showed that the effect of the fault propagation between two nodes could be effectively learned by the attention mechanism. Figure 5 also shows how our model treats different types of edges. In Fig. 5, different colors denote different edge types. The red color denotes the define-use relation, which obtains the highest proportion of occurrences of all the types of edges. For each type of edges, an attention coefficient is calculated so that one node may have multiple types of edges connecting to its neighbors, also these edges vary in thickness.

Moreover, to gain insight into how the attention mechanism treats different neighbors from an entire graph perspective, the distribution of the entropy of the attention coefficients is introduced, defined as follows:

$$H\left(\{\alpha_{e_{ij}}^{(l,m)}\}_{i\in\mathcal{N}_j}\right) = -\sum_{i\in\mathcal{N}_j}\alpha_{e_{ij}}^{(l,m)}\log\alpha_{e_{ij}}^{(l,m)}. \tag{8}$$

Intuitively, the entropy value characterizes how the uncertain node learns from its neighbors. Small entropy values indicate that the node learns heavily from a few neighbors. If we use Fig. 5 to illustrate the entropy value, small entropy values correspond to uneven attention coefficients, which means the node has edges differing in thickness significantly. A uniform distribution of attention coefficients corresponds to the highest entropy value $\log|\mathcal{N}_j|$, which means node with edges of equal attention coefficient (thickness). Due to the attention coefficients are set to the same constant value, the attention coefficients of GATPS-const follow a uniform distribution.

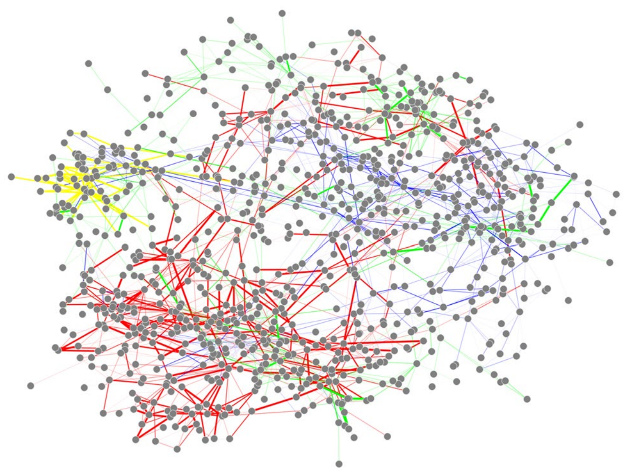


Fig. 5 Representation of attention coefficients of GATPS's attention head 0 on the *schedule2* program. Edge color denotes the type of edge (blue: branch, green: addressing, yellow: logical, red: define-use)

Figure 6 shows the entropy distribution of the eight attention heads of layer1 of the *schedule2* program. The x-axis of each subfigure shows the entropy values, and the y-axis shows the number of nodes. We introduce the entropy distribution of a uniform distribution of the attention coefficients as a baseline for other distributions. The last subfigure of Fig. 6 depicts the distribution of entropy of a uniform distribution of the attention coefficients. Compared to a uniform distribution, some attention heads (such as head 2, 5, and 7) had relatively low entropies. Thus, by applying these attention heads, the nodes learned from a few neighbors. Head 3 had a high entropy distribution, and thus, the nodes learned more equally from their neighbors than head 2, 5, 7. Two major findings could be concluded from Fig. 6. First, the distribution of attention coefficients for each attention head differed from that of assumed uniform distribution, which indicated that each attention head learned unevenly from node's neighbors. Second, the distributions of the entropies of the attention heads varied. Therefore, each attention head learned in a different manner and concentrated on various aspects of propagation. It is necessary to apply multi-head attention mechanism to make it possible to observe the various aspects of propagation. It is probable that certain attention heads reflect the intrinsic propagation semantics, such as data truncation or repeated addition [7]. The interpretation of the attention coefficients will be a subject of future work.

The F1 score of IPAS was lower than GATPS and proPV. The prediction process of IPAS was analyzed. If the instructions to be predicted were the same type of instruction and in the same bbl, the structural features might vary in only few items, incurring similar prediction results. The experiment of the benchmark *replace* showed that 99% of the *mov* instructions within the same bbl had the same prediction result. For example, *mov %esp,% ebp*, and other *mov* instructions within one bbl might obtain the same prediction result. However, *mov %esp, %ebp* affected the stack base pointer *ebp*, which likely led to the abnormal behavior when it needed to restore the return address and finally caused crash. After investigating the injection results, we found that *mov %esp,% ebp* obtained a crash rate of 100%, and other *mov* instructions obtained a crash rate of 42% and a SDC rate of 22%. Therefore, the probable reason for IPAS obtained a low F1 score is failing to discriminate between the instructions of the same type within one bbl. Due to the aggregation of neighboring nodes, our model could accurately discriminate the structural differences of the same type of instructions.

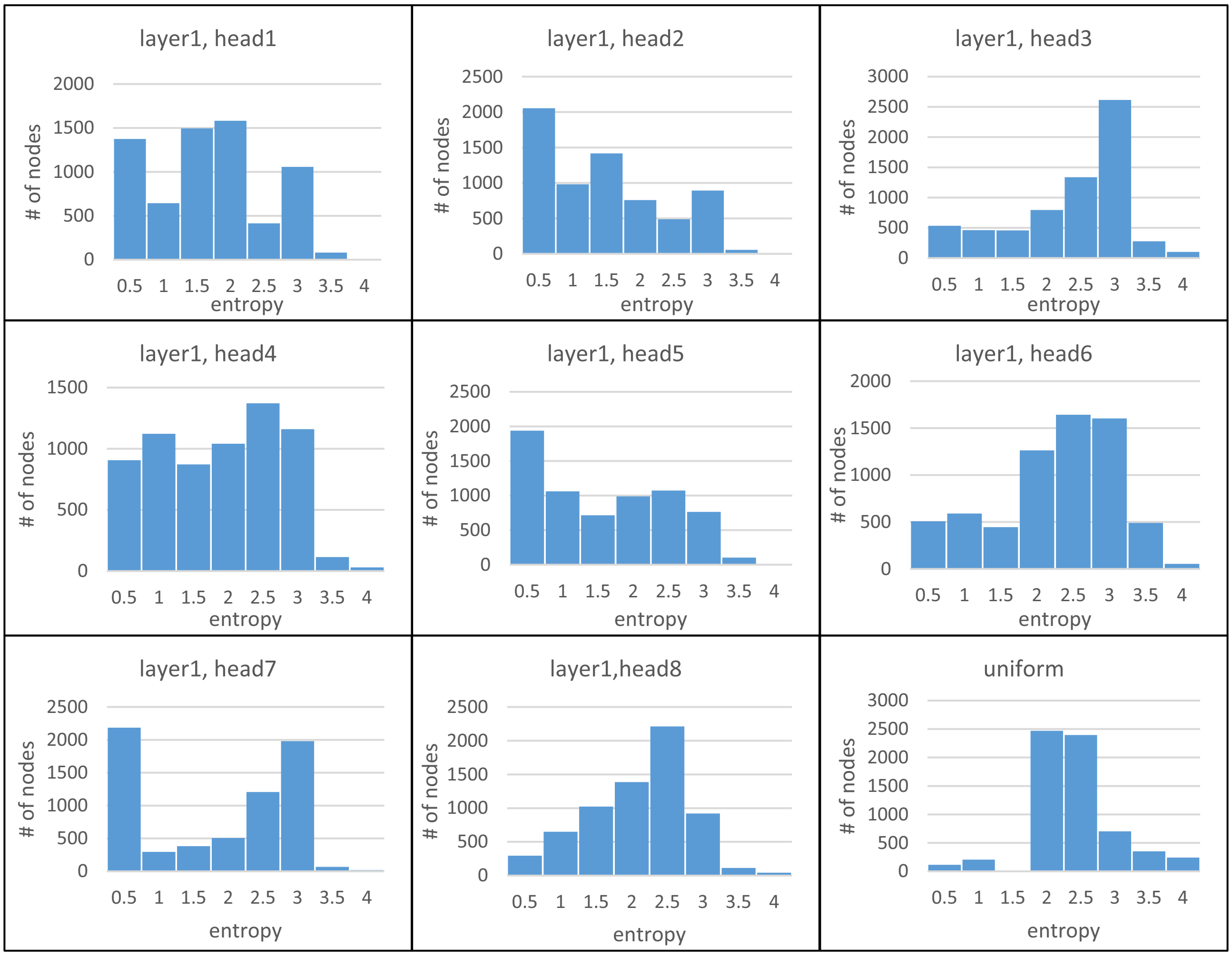


Fig. 6 Entropy distributions of eight attention heads of layer1 of *schedule2* program

proPV did not take into account the types of relations between instructions, so it could not differentiate between types of relations. proPV examined if an instruction had downstream connector instructions to propagate faults to other functions. However, it failed to further describe the patterns of propagation in which the dependent instructions were affected. As showed in Sect. 3, the effects of fault propagations varied significantly due to different relations. For example, after examining the prediction results of proPV, we found many false positive cases were because the proPV model mistakenly classified the instructions which produced data for addressing as SDC. The results of the program *tot_info* showed that 22.6% of the address-generating instructions were not correctly predicted, which incurred crash in the fault injections and were predicted as SDC-prone. As we showed in the incorrect cases, handcrafted features tended to be weak for describing the complex propagation context.

5.1.2 Time Cost

This subsection demonstrates the time costs of our model and the baseline methods. The time cost can be roughly divided into three parts: fault injection, model training, and prediction. The time cost of the prediction was trivial. In the experiment, we set the training ratio to 0.33, meaning that we needed to conduct 33% of the exhaustive fault injections to obtain the training set. Figure 7 shows the time cost and its composition. The model training process only represented 1/18 of the time compared to the fault injection process averaged across all studied programs. GATPS achieved an average 40 × speedup over the full fault injections. Each epoch required 2–12 s, and the time of each epoch was related to the scale of graph. As the program replace had the largest number of nodes, the training of replace was the most time-consuming. The training process was set to take 100 epochs. In total, the training

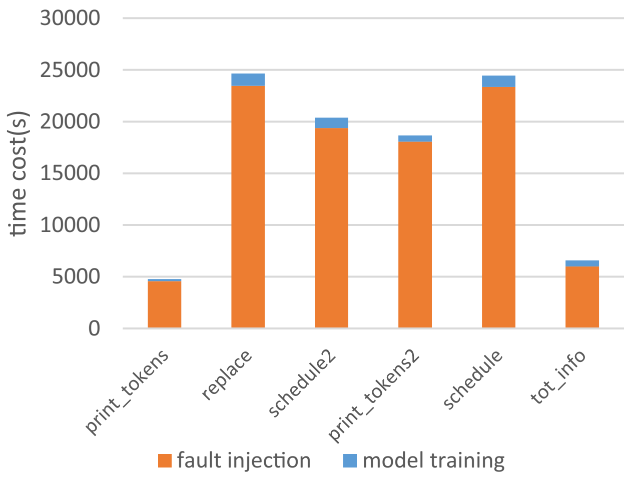


Fig. 7 Time cost of transductive learning of GATPS

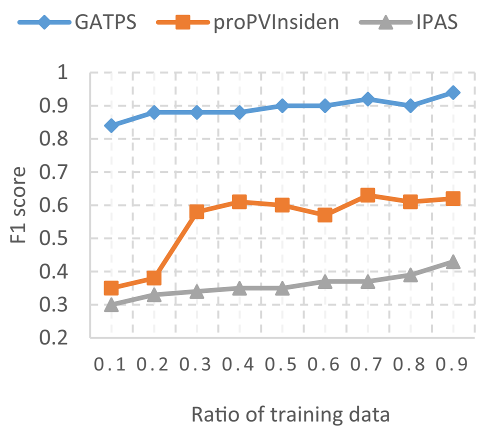


Fig. 8 Predictive performance for various training ratios

process took about 4–20 min. Since GATPS-homo model had far fewer parameters to train, the time cost of GATPS-homo was only 12% of that of GATPS.

The running time costs of proPV or IPAS for benchmark programs were less than 1 s. Compared to the time cost of conducting fault injections, the time cost of proPV or IPAS was negligible. proPV and IPAS also had much lower time costs than our model. However, because the major cost was performing fault injections, the total time cost of our model was only 5.4% larger than that of proPV or IPAS. Considering that our model obtained much more accurate (> 34%) prediction results compared to the baselines, it is worth this additional time cost.

Moreover, the training time cost could be further reduced by running the training program on a GPU. Our experiment was conducted on a CPU. We also ran model training of print_tokens on a GPU. The training of print_tokens showed that the time cost of running on a GPU was only 36.5% that of running on a CPU. However, our GPU devices, which had 10 GB of embedded memory, were not able to compute other larger scale programs due to the limited memory capacity. We attempt to compress the graph to make it able to compute GATPS on the GPUs. This is a subject of future work.

5.1.3 Parameter Sensitivity Study

In this subsection, we examine the sensitivity of the models to various parameter, including the training ratio and the number of attention layers. Figure 8 details the comparison of the F1 scores for the *schedule2* program for various values of the training ratio. The F1 scores were in the order of GATPS > proPV > IPAS. The F1 score of GATPS/proPV/IPAS varied in a small range (< 0.09) when the ratio of the training data was greater than 0.3. The F1 score of IPAS increased faster as the training ratio increased compared to other methods. Furthermore, the experiment showed that the training ratio did not affect the training time cost per epoch of GATPS. Since the F1 score remained stable and the cost of the fault injection increased linearly when the training ratio increased, it was beneficial to set the training ratio of the GATPS to a low level to balance the prediction accuracy and cost. The experiment showed that it was adequate to set the training ratio to 0.2 to obtain a stable F1 score.

We also analyzed the effect of the number of attention layers L when the other parameters were fixed. Figure 9 shows the F1 score and training time cost per epoch as L increased. The F1 score of the GATPS increased slowly as L increased. The F1 scores when $2 \leq L \leq 7$ varied in a small range (< 0.05). As the number of layers L increased, the node could learn structural information from farther neighboring nodes. The training time per epoch increased linearly as L increased. The three-layer training took 1.6 × the time of the two-layer training. For the seven-layer model, the

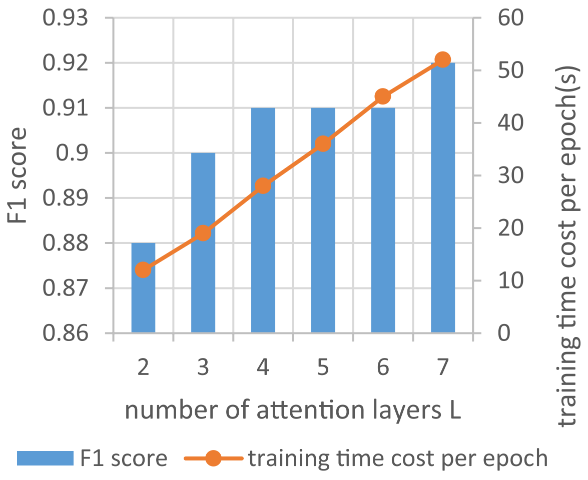


Fig. 9 Predictive performance for various numbers of attention layers

training took 4.3 × the time of the two-layer training. Compared to the rapid growth in the time cost, the increase in the F1 score was slight. The experiment showed that the two- or three-layer model provided a high prediction accuracy with a low cost, which was adequate for the general profiling needs. However, if strict goal accuracy is required, it is feasible to increase the number of attention layers. This is a trade-off that should be considered by the users.

5.2 Results for Inductive Learning

In this subsection, we quantify the prediction performance of inductive learning. Different from transductive learning, inductive learning predicts SDC-prone instructions without any fault injection information of the target program, which requires a higher generalization ability. We performed two experiments to show how the input graph affected the prediction performance.

1. N-inputs. The first experiment applied the graphs generated from the execution of the same program with different inputs (N-inputs). Although the executions with different inputs varied in data flow and control flow, they may have the same functions or bbls, resulting in similar graphs.
2. N-programs. The second experiment applied the graphs generated from the execution of totally different programs (N-programs). The results of the two experiments are presented separately.

5.2.1 Model Trained by N-inputs

The dataset contained six graphs for training and two for validation. After training the inductive model, we performed predictions on three test inputs separately. These graphs were generated from the executions of 11 distinct inputs for the program of *schedule2*. The testing graphs remained completely unobserved during training.

Prediction Performance Table 6 shows the comparison of the results of N-inputs with the baselines. Our model achieved a higher precision, recall and F1 score than baselines. The F1 score of our model was 54% higher than that of IPAS and 37% higher than that of proPV, showing that our model had a higher ability to predict unseen graphs. Compared to transductive learning, the differences between the inductive learning and baseline results were larger because inductive learning required a higher ability of generalization.

The comparison of the inductive learning results between the GATPS and GATPS-homo models showed that applying a heterogeneous graph was beneficial. The average F1 score of GATPS was 7% higher than that of GATPS-homo. This conclusion is consistent with the transductive learning results.

Table 6 Comparison of results for SDC prediction of inductive model trained by N-inputs

Metrics		P	R	F1
GATPS	test1	0.86	0.78	0.82
	test2	0.80	0.86	0.83
	test3	0.88	0.69	0.77
	average	0.85	0.78	0.81
GATPS-homo	test1	0.89	0.74	0.80
	test2	0.73	0.70	0.71
	test3	0.90	0.60	0.72
	average	0.84	0.68	0.74
IPAS	test1	0.19	0.49	0.27
	test2	0.14	0.44	0.21
	test3	0.22	0.54	0.32
	average	0.18	0.42	0.27
proPV	test1	0.52	0.46	0.49
	test2	0.31	0.38	0.34
	test3	0.72	0.38	0.49
	average	0.52	0.41	0.44

Time Cost The average training time per epoch was 93 s, which was 9.3 × the time cost of transductive learning. This was because the scale of the model increased. The training input of the transductive learning was one graph, and the input of the inductive learning increased to six graphs. The time cost of the GATPS for predicting SDC-prone instructions under a single input was only 6.8% that of the exhaustive fault injections.

Furthermore, we could repeatedly use the trained model of the GATPS to predict the SDC-prone instructions for different inputs. The function of keras *load_weights* could be used to reload the trained weights of each attention layer. Thus, the subsequent predictions did not require training. If we needed to perform predictions for *n* inputs, the time cost of the GATPS could be amortized over *n* tests, and each prediction would require only *1/n* of the training time cost.

Parameter Sensitivity Study We discuss the influence of the parameters of the inductive learning model, including the number of layers and the training ratio. Figure 10 shows the effect of the number of attention layers L when the other parameters were fixed. The results of the GATPS and GATPS-homo training by N-inputs are shown. The maximum number of attention layers was set to 7. The F1 score increased by 9% when L increased from 2 to 7. Figure 10 also shows the training time cost per epoch. The training time increased linearly as the number of layers increased. For the seven-layer model, the training required 5.2 × the time required for the two-layer training. The 2-layer model obtained a high F1 score with low lost, which was the most

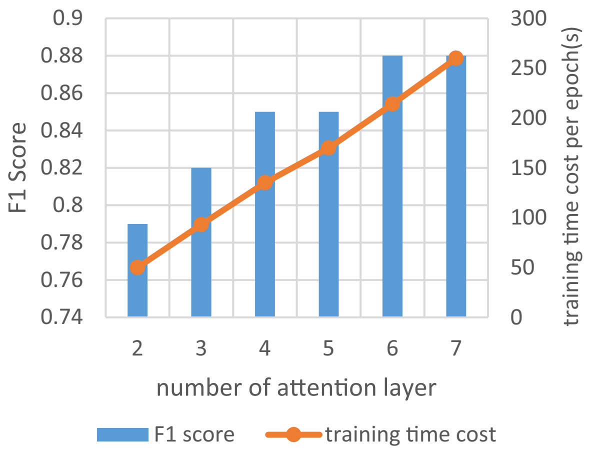


Fig. 10 Predictive performance for N-inputs inductive learning for various numbers of attention layers

cost-effective compared to other numbers of layers. When L increased from 2 to 3, the F1 score increased by 3.8% while the time cost increased by 86.0%.

Figure 11 shows the result comparison of the F1 scores on the *schedule2* program, varying the number of graphs in the training set. The validation set and test set remained unchanged during the experiment. The F1 score increased when the number of graphs increased, and it varied in a small range (< 0.05) when the number of graphs ≥ 4. The training time cost per epoch increased linearly as the number of graphs in the training set increased. When the number of graphs was nine, the training time cost increased to 7.1 × the time cost with one graph in the training set. According to the result of experiment, it was adequate to set the number of graphs in the training set to 4 to obtain a stable F1 score for model trained by N-inputs.

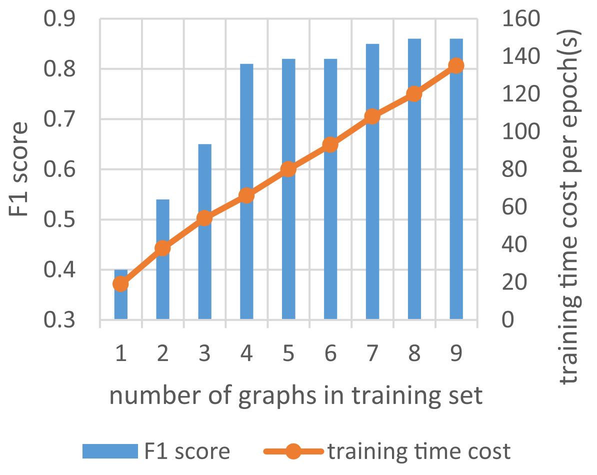


Fig. 11 Predictive performance of N-inputs inductive learning for various numbers of graphs in the training set

5.2.2 Model Trained by N-programs

The dataset contained four graphs for training, one for validation, and three for testing. The training set was the dataset of four distinct programs {*schedule*, *print_tokens*, *print_tokens2*, and *tot_info*}, the validation set was {*replace*}, and the test set was three distinct programs {*schedule2*, *kmp*, and *dfs*}. The functionalities of the training and validation sets were totally different from the test set, so the test set was completely unseen.

Prediction Performance Table 7 shows the comparison of the results of our models trained by N-programs with baselines. The average F1 score for inductive learning trained by N-programs was 0.59. The results validated that the GATPS could learn certain general propagation knowledge and accommodate unseen programs. The result for the same test samples of *schedule2* showed that the F1 score was 16% lower than that of the model trained by N-inputs. Thus, the input graphs affected the prediction performance significantly. The model trained by N-inputs could learn application-specific knowledge, but the model trained by N-programs could not. The difference between the F1 scores indicated that certain SDC propagations were application specific, and thus, the model trained by N-inputs could achieve a higher F1 score.

The F1 score of GATPS-homo was 3% lower than that of GATPS, which was consistent with the result of transductive learning. Moreover, IPAS and proPV obtained much lower F1 scores than GATPS. The F1 scores of IPAS and proPV were 27% and 38% lower than that of GATPS. Notably, the inductive

Table 7 Comparison of results for SDC prediction of inductive learning trained N-programs

Metrics	Test program	P	R	F1
GATPS	*schedule2*	0.64	0.67	0.66
	kmp	0.57	0.48	0.52
	dfs	0.86	0.45	0.59
	average	0.69	0.53	0.59
GATPS-homo	*schedule2*	0.42	0.81	0.55
	kmp	0.46	0.61	0.53
	dfs	0.78	0.48	0.59
	average	0.55	0.63	0.56
IPAS	*schedule2*	0.13	0.76	0.22
	kmp	0.16	1.00	0.28
	dfs	0.29	1.00	0.45
	average	0.19	0.92	0.32
proPV	*schedule2*	0.14	0.30	0.19
	kmp	0.12	0.06	0.08
	dfs	0.38	0.36	0.37
	average	0.21	0.24	0.21

learning of GATPS achieved a higher F1 score than the transductive learning of IPAS and proPV, showing that our model had a better ability for reasoning about fault propagation.

Time Cost The average training time per epoch was 53 s, which was 5.3 × the time cost of transductive learning. The time cost was less than the time cost of the training by N-inputs because the training set was smaller. The GATPS trained by N-programs achieved an average 43 × speedup over the full fault injections. As we stated before, the time cost could be roughly divided into three parts: fault injection, model training, and prediction. The model trained by N-programs did not require information about the target program, and the fault injection and model training could be performed in advance. If we knew the target program, we would only need to load the trained model to predict the SDC-prone instructions. Thus, the time cost that affected the user was only the prediction time cost.

Parameter Sensitivity Study In this section, we discuss the influence of the number of layers, which is shown in Fig. 12. The F1 score changed in a larger range as the number of layers increased compared to transductive learning and N-inputs training. The largest difference between the maximum and minimum F1 scores was 0.21. For comparison, the largest differences were 0.09 (N-inputs) and 0.04 (transductive learning). Moreover, the model trained using more attention layers did not show a higher F1 score, which was also different from the results of transductive learning and N-inputs training. The models with varying attention layers may concentrate on different aspects of the fault propagation. Since the test program was never observed, more attention layers did not cause the trained model to predict more fault propagations of the test program correctly. However, the training time increased linearly as the number of layers increased. The three-layer training took 1.8 × the time of the two-layer training. For the seven-layer model, the training took 4.2 × the time of the two-layer training. More attention layers did not bring a higher prediction accuracy, and thus, we can apply the simple two-layer model to reduce the time cost in practice.

5.3 Discussion

In this subsection, we discuss the predictive results of the GATPS (transductive learning, inductive learning trained by N-inputs and N-programs, and homo graph model). Figure 13 shows the prediction results for *schedule2* by the GATPS models. We discuss the F1 scores and time costs of these models.

1. F1 scores: transductive learning > inductive learning (N-inputs) > inductive learning (N-programs)

   To explain this result, we broadly divide the propagation knowledge into general knowledge and application-specific knowledge. General knowledge refers to the propagation schemes that suit all programs. For example, if a fault occurs in a dynamically dead instruction, it definitely results in a benign outcome [4]. Furthermore, many propagations are related to program-specific knowledge. For example, faults in the memory address or branch may cause different behaviors in different programs [16]. The model trained by N-programs can learn only general knowledge, and the model trained by N-inputs can learn both general knowledge and application-specific knowledge. Transductive learning directly learns propagation knowledge from the target program, and its application-specific knowledge is even more precise than that trained

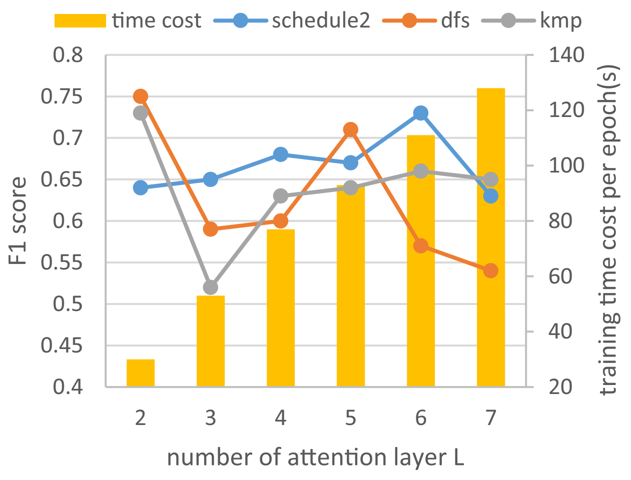


Fig. 12 Predictive performance of N-programs inductive model for various numbers of attention layers

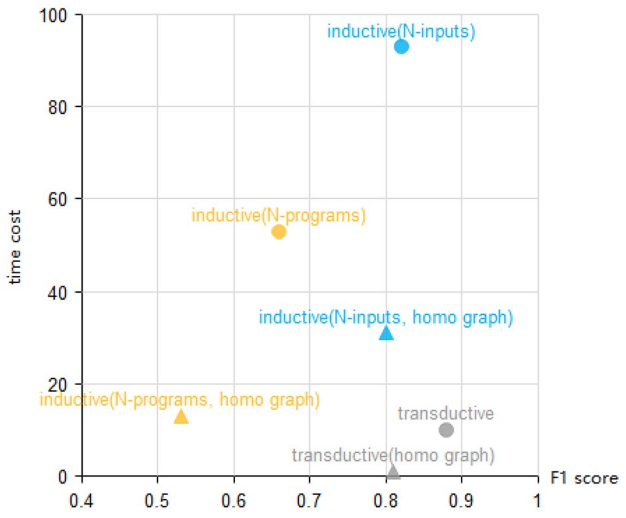


Fig. 13 Comparison of the predictive performances of different models

by N-inputs since different inputs still vary in execution details.

2. Time cost: inductive learning > transductive learning

   The time cost of the model is largely related to the scale of the input graphs. The scale of the input graphs of inductive learning is usually larger than that of transductive learning, i.e., the input of transductive learning is one graph and the input of inductive learning is multiple graphs, so inductive learning often has a higher training cost. However, the time cost of inductive learning can be amortized over multiple tests. Once the model is constructed, we can predict SDC-prone instructions over other inputs or even programs. Furthermore, training of inductive learning can be performed before we know the target program, and the trained model can be loaded immediately when it needs to predict SDC-prone instructions of the target program. For transductive learning, after we know the target program, we have to conduct fault injections to acquire the labels of the training set of the target program and perform model training. Therefore, although inductive learning has a higher total time cost than transductive learning, it can provide prediction results in a shorter time than transductive learning after user provides the target program.

To conclude, we summarize the characteristics of the transductive learning, inductive learning, and homo graph models of the GATPS.

- The advantage of transductive learning is a high prediction accuracy. The prediction of the SDC-prone instructions achieved the highest F1 score. The disadvantage is that it requires a portion of the fault injection results of the target program before training.
- The advantage of inductive learning is that it does not need to perform injections to the target program, which enables the decoupling of the vulnerability assessment from the fault injections. However, its accuracy is lower than that of transductive learning.
- The advantage of the homo graph model is a lower time cost compared to the heterogeneous graph model. It requires only 12%–33% time of the heterogeneous graph model, and its F1 score is 3%–12% lower than those of the heterogeneous graph model.

To satisfy user demand, different GATPS models can be used based on the time cost and data provided. We discuss some typical application scenarios.

- If fault injection cannot be conducted on the target program, we can apply an inductive learning model of the GATPS. The trained model of N-programs can be loaded to make predictions.
- If SDCs under multiple inputs of one program need to be predicted, we can apply an inductive model trained by N-inputs and predict SDC-prone instructions.
- If the time cost constraint is strict and the heterogeneous graph model of the GATPS cannot satisfy the requirement, the homo graph model can be used to reduce the time cost.

6 Conclusion

This paper predicts SDC-prone instructions by using a heterogeneous graph attention network. To obtain the context of fault propagation, the embedding of each instruction is learned by aggregates of its neighbors' information. The self-attention mechanism is applied to evaluate the influences of errors in neighboring nodes. Our approach is different from previous machine learning approaches since we perform SDC prediction in an end-to-end manner instead of using handcrafted features. Experimental results showed that our model outperformed previous models applying handcrafted features. We also constructed an inductive model to predict SDC-prone instructions without any fault injections on target programs, minimizing the dependence on the fault injections. The advantages, disadvantages, and typical application scenarios of transductive learning and inductive learning of the GATPS models were discussed to provide solutions for different vulnerability assessment demands.

Our future work will focus on interpretation of the graph model. Our model learns propagation knowledge to predict SDC-prone instructions. However, this knowledge is expressed implicitly, which is not readable for researchers. It is helpful to interpret the attention coefficients to gain insights into the general knowledge of fault propagation. Moreover, by using the instruction embeddings obtained in this paper we can compute the embeddings for higher levels (basic block, function or program) to solve other fault propagation problems such as computing probability of a corrupt branch affects final output or computing overall SDC probability for the program.

Funding This work was funded by the Natural Science Foundation of China (No.62002030), Key research and development plan project of the Shaanxi Province, China (No.2019ZDLGY17-08, 2019ZDLGY03-09–01, 2019GY-006, 2020GY-013).

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

Competing Interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Abadi M, Barham P, Chen J et al (2016) Tensorflow: A system for large-scale machine learning. In: Proc. USENIX symposium on operating systems design and implementation (OSDI). IEEE, pp 265–283
2. Benacchio T, Bonaventura L, Altenbernd M et al (2021) Resilience and fault tolerance in high-performance computing for numerical weather and climate prediction. Int J High Perform Comput Appl 35(4):285–311
3. Dixit HD, Pendharkar S, Beadon M et al (2021) Silent data corruptions at scale. arXiv preprint. http://arxiv.org/abs/2102.11245
4. Fang B, Lu Q, Pattabiraman K et al (2016) ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In: Dependable Systems and Networks (DSN). IEEE, pp 168–179
5. Gao Y, Gupta SK, Wang Y et al (2014) An energy-aware fault tolerant scheduling framework for soft error resilient cloud computing systems. In: Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1–6
6. Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, pp 249–256
7. Guo L, Li D, Laguna I (2021) Paris: Predicting application resiliency using machine learning. J Parallel Distrib Comput
8. Hashimoto M, Wang L (2020) Soft error and its countermeasures in terrestrial environment. In: Proc. Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, pp 617–622
9. Hari SKS, Adve SV, Naeimi H (2012) Low-cost program-level detectors for reducing silent data corruptions. In: Dependable Systems and Networks (DSN). IEEE, pp 1–12
10. Hari SKS, Adve SV, Naeimi H et al (2012) Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, pp 123–134
11. Hamilton W, Ying Z, Leskovec J (2017) Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems (NIPS). IEEE, pp 1024–1034
12. Hong H, Guo H, Lin Y et al (2020) An attention-based graph neural network for heterogeneous structural learning. In: Proc. Conference on Artificial Intelligence (AAAI). AI Access Foundation, pp 4132–4139
13. Kalra C, Previlon F, Rubin N et al (2020) Armorall: Compiler-based resilience targeting gpu applications. ACM Trans Archit Code Optim 17(2):1–24
14. Laguna I, Schulz M, Richards DF et al (2016) Ipas: Intelligent protection against silent output corruption in scientific applications. In: Proc. International Symposium on Code Generation and Optimization (CGO). IEEE, pp 227–238
15. Li G, Pattabiraman K (2018) Modeling input-dependent error propagation in programs. In: Dependable Systems and Networks (DSN). IEEE, pp 279–290
16. Li G, Pattabiraman K, Hari SKS et al (2018) Modeling soft-error propagation in programs. In: Dependable Systems and Networks (DSN). IEEE, pp 27–38
17. Li Z, Menon H, Maljovec D et al (2020) SpotSDC: Revealing the silent data corruption propagation in high-performance computing systems. IEEE Trans Vis Comput Graph
18. Li Z, Menon H, Mohror K et al (2021) Understanding a program's resiliency through error propagation. In: Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM, pp 362–373
19. Liu C, Gu J, Yan Z et al (2019) SDC-causing error detection based on lightweight vulnerability prediction. In: Proc. Asian Conference on Machine Learning (ACML). IEEE, pp 1049–1064
20. Lu Q, Pattabiraman K, Gupta MS et al (2014) SDCTune: A model for predicting the SDC proneness of an application for configurable protection. In: Compilers, Architecture and Synthesis for Embedded Systems (CASES). ACM, pp 1–10
21. Luk CK, Cohn R, Muth R et al (2005) Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices 40(6):190–200
22. Ma J, Wang Y (2017) Characterization of stack behavior under soft errors. In: Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1538–1543
23. Maaten L, Hinton G (2008) Visualizing data using t-SNE. J Mach Learn Res 9:2579–2605
24. Schlichtkrull M, Kipf TN, Bloem P et al (2018) Modeling relational data with graph convolutional networks. In: Proc. European Semantic Web Conference. Springer, pp 593–607
25. Velickovic P, Cucurull G, Casanova A et al (2018) Graph attention networks. In: Proc. International Conference on Learning Representations (ICLR). IEEE, pp 1–12
26. Xin X, Li ML (2012) Understanding soft error propagation using efficient vulnerability-driven fault injection. In: Dependable Systems and Networks (DSN). IEEE, pp 1–12
27. Yang N, Wang Y (2019) Predicting the silent data corruption vulnerability of instructions in programs. In Proc. International Conference on Parallel and Distributed Systems (ICPADS). IEEE, pp 862–869

Junchi Ma received the Ph.D. degree in computer science and engineering from Southeast University, China, in 2017. He is currently working in the school of information engineering, Chang'an University, P. R. China. His research interests include software reliability and graph representation learning.

Zongtao Duan received his Ph.D. in computer science from Northwestern Polytechnical University, P. R. China in 2006. He is currently a professor at the school of information engineering, Chang'an University, P. R. China. He was a postdoctoral research fellow in the University of North Carolina, American in 2009-2010. His research interests include context-aware computing in transportation. He is a member of CCF, CCF High Performance Computing, and Pervasive Computing Technical Committee.

Lei Tang received the Ph.D. degree in computer science and technology in 2012. She is currently working in the school of information engineering, Chang'an University, P. R. China. She was a visiting researcher at the chair of information systems, Mannheim University, Germany in 2009-2010. Her research interests include graph representation learning and intelligent transportation system. She is a member of ACM, IEEE, and CCF (China Computer Federation).