README

We provide this README file to illustrate in detail on the implementation process of GraphFI.

**Setup:**

We deploy GraphFI on a host Intel i7-7700 CPU (2.60 GHz) with 16 GB memory and an NVIDIA GeForce RTX 3090 GPU. We use a cycle-accurate, open-source simulator GPGPU-sim to obtain the dynamic instruction instances of each thread(i.e., the whole fault space).

**Usage：**

Below, we provide a detailed description of the implementation of GraphFI, including the following four steps: 1. ID-GraphFI, 2. TD-GraphFI, 3. MD-GraphFI, and 4. Fault Injection.

**1.ID-GraphFI:**

We first modified the output options of **GPGPU-sim** to output the **dynamic instruction set** along with the **corresponding registers**. Then, we filtered the address calculation instructions from the dynamic instruction set to estimate the proportion of address calculations, and finally ran **"kernel.cuh"** to obtain the active vertex set. We check the neighbor state loading operation ratio of different iterations. As this ratio typically remains stable, we randomly select individual iteration to represent the overall Detected rate of the program. Then we divide consecutive iterations with similar active vertex sets into an iteration group, and select a representative iteration from each group as FI candidate to efficiently evaluate the program SC rate.

**2.TD-GraphFI:**

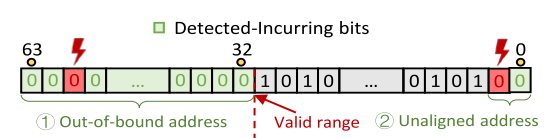
After selecting representative iterations, we utilize the correlation between vertices' error re-silience and their neighborhood similarity/clustering properties to identify vertex communities in which vertices share similar neighbors (N-Communities) or (b) in highly clustered subgraphs (C-Communities). From each set, we select one representative vertex for fault injection.

Running the **"similar.py"** program generates the neighbor similarity set (N-Communities).

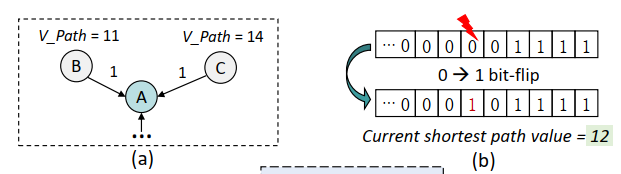
Running the **"cluster.py"** program generates the high-clustered set (C-Communities).

**3.MD-GraphFI:**

For each vertex (thread) that executes many instructions, for address calculation instructions, the base address of the array can be obtained (such as through a pointer variable in CUDA). Then, the valid address range can be estimated based on the size of the array elements and the thread's index offset (calculated by thread ID and block ID), which allows us to identify the higher bits that may cause errors. We obtain the register information through GPGPU-sim, and based on this valid address, we can estimate which bits in the address register will lead to detected errors, as shown in the figure below.



Due to certain graph update operations have fault masking properties, as shown below, such as in SSSP, where the min() function selects the shortest path from all in-edge neighbors and updates the target vertex path accordingly. Faults in the higher bits of non-shortest path operands may be masked by the min() function.



We first exclude the “abnormal” bits whose error-incurred outcome is surely BC according to the fault masking property of the graph updating operation (if there is). The remaining faults can be handled by binary search fault injection based on monotonicity."

**4.Fault Injection:**

Based on the above analysis, we perform fault injection experiments. Running the script **"inject\_one.sh"** allows for a specified number of fault injections. The main file for fault injection is **"fault-inject.py"**, where faults can be injected into **representative threads and instructions** according to the above analysis and corresponding configuration of the fault injection script. The outputs of the fault injection are **"outcome.txt"** and **"basic.txt"**, which record the results of fault injection (i.e., BC, SC, and Detected) as well as detailed information for each fault injection experiment.