Resilience Assessment of Large Language Models under Transient Hardware Faults

Udit Kumar Agarwal, Abraham Chan, Karthik Pattabiraman
The University of British Columbia , Canada.
Email: uditag97@student.ubc.ca, abrahamc@ece.ubc.ca, karthikp@ece.ubc.ca

***Abstract*—Large Language Models (LLMs) are transforming the field of natural language processing and revolutionizing the way machines interact with humans. LLMs like ChatGPT and Google's Bard have already made significant strides in conversational AI, enabling machines to understand natural language and respond in a more human-like manner. In addition to typical applications like sentiment analysis and text generation, LLMs are also used in safety-critical applications such as code generation and speech comprehension in autonomous driving vehicles, where reliability is important.**

In this work, we investigate the resilience of LLMs under transient hardware faults. Specifically, we used IR-level fault injection (FI) to assess the reliability of five popular LLMs, including Bert, GPT2, and T5, under transient hardware faults. Moreover, we also investigate how the resilience of LLMs varies with different pre-training, fine-tuning objectives, and the number of encoder and decoder blocks. We find that LLMs are quite resilient to transient faults overall. We also find that the behavior of the LLM under transient faults varies significantly with the input, LLM's architecture, and the type of task (e.g., translation vs. fill-in-the-blank). Finally, we find that the Silent Data Corruption (SDC) rate varies with different fine-tuning objectives, and for the fill-mask fine-tuning objective, the SDC rate also increases with the model size. Overall, our findings indicate that the use of LLMs in safety-critical applications needs further investigation.

***Index Terms*—Error resilience, LLMS, Soft Errors.**

I. Introduction

Large Language Models (LLMs) are redefining the possibilities of natural language processing (NLP), and reshaping the way we interact with machines. With their ability to generate natural-sounding language and understand the context and meaning of words and sentences, LLMs are redefining what machines can do with language. A recent LLM by OpenAI, ChatGPT [1], broke all records by gaining 100 Million users in just two months of its release, and is widely used by both the tech-savvy and the general public.

Along with typical applications like sentiment analysis, text summarization, and generation, LLMs are also used in safety-critical applications like code generation (Github Copilot [2], CodeBert [3]), hazard analysis of Autonomous Braking Systems [4], language translation, and speech comprehension in Autonomous Driving Vehicles (ADVs) [5]. Moreover, the advent of multi-modal LLMs (like GPT4) is opening up new possibilities for LLMs in safety-critical applications, particularly in the field of autonomous driving [6]. For example, Mercedes [7] and General Motors [8] recently announced the inclusion of LLMs like ChatGPT in their ADVs to enable voice commands and natural language interactions between the vehicle and passengers. The rationale is that this allows for more intuitive and flexible communication, enhancing the user experience inside the ADV. Therefore, it is important to ensure the reliability of LLMs.

In this paper, we experimentally investigate LLMs' resilience under *transient hardware faults (i.e., soft errors)* in the processor. Transient hardware faults are induced by several factors, such as cosmic rays, electromagnetic interference, power surges, and other environmental circumstances [9], resulting in short-lived disruptions in the hardware components of a system. These faults are increasing in frequency due to the effects of technology scaling and cannot be masked by hardware due to power constraints [10].

Transient hardware faults can cause the software to either crash or result in a Silent Data Corruption (SDC), thus causing the software to silently produce the wrong output. SDCs can cause ML models to incorrectly classify objects and make false predictions, which can be problematic in safety-critical applications. For instance, in ADVs, transient hardware faults can cause the object detection model to misclassify objects, potentially resulting in collisions [11]. They may also violate the safety standards of many of these applications, e.g., ISO26262 [11], which specifies the maximum number of failures allowed under hardware and software faults. Therefore, in this paper, we focus on SDCs due to transient hardware faults.

Prior work has evaluated the effects of transient hardware faults on Deep Neural Networks (DNNs [11]) and Recursive Neural Networks (RNNs [12]), which are different from LLMs. Other work has explored other dimensions of LLM's reliability (and security) under adversarial attacks [13], noise in input prompts [14], and out-of-distribution user inputs [15], but not under hardware faults. *To the best of our knowledge, we are the first study to experimentally evaluate the resilience of LLMs under transient hardware faults.*

In this work, we use LLTFI [16], an LLVM [17] Intermediate Representation (IR)-level fault injector, to emulate transient hardware faults during inference of five popular LLMs: BioBert, CodeBert, T5, Roberta, and GPT2. LLTFI works by injecting the fault (e.g., single bitflip) in the results of randomly-selected instructions of the ML program at runtime. To integrate LLMs with LLTFI, we made two modifications to it. First, we added an auxiliary FI runtime that statically links with the LLM executable. Our FI runtime

has low runtime overhead, thus allowing LLTFI to scale to LLMs[1]. Second, we improved LLTFI's ML fault propagation tracing to scale to LLMs, thereby allowing us to dive deep into the causes of the SDCs observed in our experiments.

We then perform extensive fault injection (FI) campaigns to measure the SDC rate (i.e., the percentage of undetected incorrect outputs) of LLM benchmarks. Moreover, we also present a qualitative analysis of SDCs in LLMs by categorizing the SDCs based on their syntactic or semantic (in)correctness. Finally, we also evaluate how the SDC rate varies with FI in different layers of LLMs, the bit positions, the type of pre-training, and fine-tuning objectives.

Based on our experiments, we found that LLMs are quite resilient under transient faults, with the average SDC rate being 0.9% (9 SDCs out of 1000 FI experiments) across all benchmarks. Also, the manifestation of SDCs varies significantly with the type of LLMs and fine-tuning objectives. For example, for T5, an LLM for language translation, we found several interesting manifestations of SDCs, where the corrupted output semantically differs from the correct one. Finally, we observed that the SDC rate varies with different fine-tuning objectives, and for the fill-mask fine-tuning objective, the SDC rate increases with the model size.

II. Background and Fault Model

In this section, we explain our fault injection tool, our fault model, and provide a brief background on LLMs, as necessary for understanding this work.

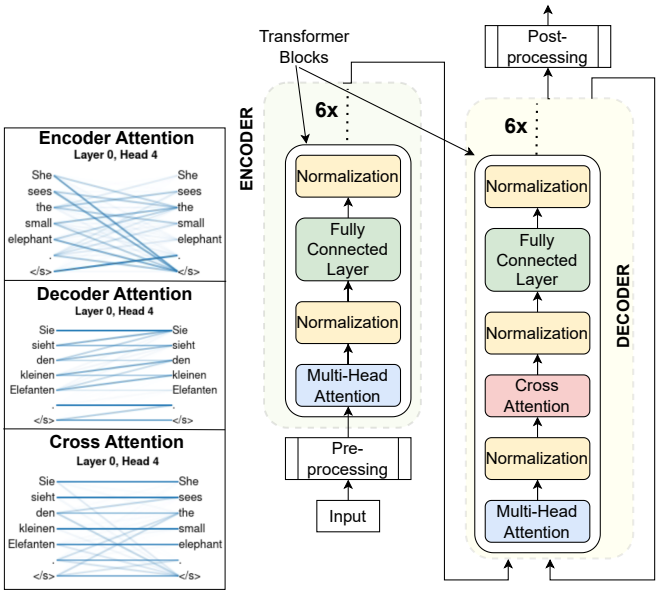
A. Large Language Models

Large Language Models (LLMs) are deep learning models trained on massive amounts of text data to learn the statistical patterns in language and generate human-like text. Once pre-trained, these models are fine-tuned on specific downstream tasks, such as text classification, sentiment analysis, or language generation.

1) Transformer-based Models: Transformer-based models, first introduced by Vaswani et al. [19], are a type of neural network architecture that has revolutionized the field of NLP. Unlike RNNs and LSTMs, Transformer-based models excel at capturing dependencies between different parts of the input sequence in a parallel manner. The key component of a transformer model is the attention mechanism:

a) ***Attention Mechanism****:* It refers to a way for the LLM to assign varying degrees of importance (called attention weights) to different tokens in the sequence. Intuitively, it allows the model to attend to specific parts of the input that are deemed more relevant for the current context. In practice, a transformer block incorporates multiple instances of these attention mechanisms (called *Multi-head attention*), running simultaneously while grasping various facets of the input sequence. In some transformer models, like T5 [20] (See Figure 1b, layer highlighted with Red), we also have *Cross attention* layers that effectively align and transfer information from the encoder to the decoder.

[1]All our code modifications and experimental data are available at: https://doi.org/10.5281/zenodo.8267719.



(a) Attention weights. (b) Architecture of T5 model.

Fig. 1: Attention weights and architecture of T5. Attention weights correspond to the fifth head of the first encoder, decoder, and cross-attention layer of a fully-trained T5 model, trained to translate English to German. The attention weights are visualized using BertViz [18].

Figure 1a shows the attention weights of the fourth head of the first encoder, decoder, and cross-attention layer of the T5 [20], English-to-German translation model. Self-attention is visualized through connecting lines between attending tokens (on the left) and the tokens being attended to (on the right). The thickness of the lines represents the attention score: thick lines indicate a larger attention score between the token on the left and that of the right. In the cross-attention layer, notice how the model has formed thicker connections between words in English and their counterparts in German.

As shown in Figure 1b, a transformer block consists of a multi-head attention layer, followed by a normalization layer and a fully-connected layer. Multiple transformer blocks (six in the case of T5) are stacked to form encoders and decoders.

b) ***Encoders****:* are responsible for processing the input sequence, such as a sentence, paragraph, or document, and capturing the contextual representations of each word or token. An encoder generates representations of the input sequence that capture its semantics and contextual information.

c) ***Decoders****:* take the encoded representations generated by the encoder and use them to generate output sequences for tasks e.g., translations, summaries, completions.

2) Different architectures of transformer-based LLMs: There are three different architectures of transformer-based LLMs:

a) ***Encoder-Only (Auto-Encoding)****:* As the name suggests, these transformer models consist only of the encoder

component and do not include a decoder. They are often used in scenarios where the primary goal is to capture the semantic and contextual information from the input sequence. Encoder-only transformers have proven helpful in various downstream tasks, including text classification, sentiment analysis, document retrieval, and feature extraction. Popular Encoder-only transformers are Bert [21] and RoBerta [22].

b) ***Decoder-Only (Auto-Regressive):*** As opposed to Encoder-only models, Decoder-only transformers are trained in an autoregressive manner, where the model learns to predict the next token in the sequence given the previous tokens. During training, the model maximizes the likelihood of generating the correct output sequence. Models like GPT2 [23], and its progeny ChatGPT, lies in this category.

c) ***Encoder-Decoder:*** These models, also known as sequence-to-sequence models, consists of both an encoder and a decoder. They are widely used in various NLP tasks, including machine translation, text summarization, and dialogue generation. Models like T5 [20] and BART [24] have an encoder-decoder architecture.

3) Input, Output Processing, Pre-training Objectives:

a) ***Pre-training objectives:*** Pre-training objectives can be supervised, unsupervised, or a combination of the two. Unsupervised pre-training objectives include de-noising objectives that involve introducing noise or corruption to the input data and training the model to reconstruct the original, uncorrupted data. Common de-noising unsupervised pre-training objectives include Masked-Language Modeling (randomly masking tokens in the input text) and Denoising Autoencoding (corrupting input text by random token dropout, token replacement, etc.). Contrary to unsupervised pre-training objectives, supervised pre-training objectives utilize labeled data to guide the model's learning process. Sentiment analysis, Named Entity Recognition (NER), and Text classification are examples of supervised pre-training objectives. LLMs like T5 are trained on unsupervised and supervised pre-training objectives [20]. While unsupervised training helps LLMs capture general linguistic knowledge and understand language structure, supervised training enables them to fine-tune their representations for specific tasks and achieve higher performance on those tasks.

b) ***Tokenization:*** Tokenization refers to the process of breaking down the input text into smaller units called tokens. It enables the models to handle variable-length input and facilitates standardization and normalization of the text, ensuring consistent handling of punctuation, capitalization, and variations. LLMs commonly use sub-word tokenization that consists of breaking down a word into sub-word units to identify its root, prefix, and suffix. For instance, the word "Reliability" gets broken down into two tokens: "Reliable" (root) and "ity" (suffix). This allows LLMs to handle verbs, nouns, plural and compound words.

Tokenization also requires a vocabulary file as input that acts as a lookup table to map tokens to unique numerical IDs. It defines the set of tokens used by the LLM and enables the model to represent and process text as numerical values.

B. LLTFI: Fault Injection Framework

We use the publicly-available LLTFI injector developed in our group in our experiments. LLTFI (Low-level Tensor Fault Injector), the successor of LLFI, is an LLVM-IR level fault injector [17] that supports injecting transient hardware faults in C/C++ and machine learning applications. LLTFI enables injecting faults in individual instructions and registers and is thus more accurate, in terms of its ability to emulate transient hardware faults [16], than application-level fault injectors like TensorFI [25]. Moreover, LLTFI provides fault-propagation tracing support for ML applications.

LLVM IR is the intermediate representation used within the LLVM Compiler Infrastructure. It follows a static-single assignment (SSA) format [26] that is designed to be hardware-independent. LLVM IR has its own instruction set that provides a structured representation of the program's control flow, data types, and operations. Moreover, LLVM provides APIs to transform LLVM IR, and LLTFI uses these APIs to identify specific instructions and carry out the FI.

LLTFI works by first converting ML models into the ONNX format [27], which is an open-source format designed to enable interoperability between different deep learning frameworks. Note that the conversion of the ML model to the ONNX format is semantics preserving as LLTFI, by default, does not use any ONNX graph optimizations [16]. The ONNX file is then converted into LLVM IR using the ONNX-MLIR [28] tool. After this, LLTFI modifies the LLVM IR to inject faults in instructions and registers. We chose LLTFI for our experiments primarily because of its framework-agnostic nature: LLTFI can work with ML models in multiple ML frameworks like Keras [29], PyTorch [30], Tensorflow [31].

C. Fault Model

In this work, we primarily consider transient hardware faults in the processor (CPU). Transient hardware faults occur due to particle strikes on hardware elements such as flip-flops, thus resulting in bit-flips from 0 to 1 or vice versa [9], in the registers and data path of the CPU. We used the single-bit flip model to represent transient faults in hardware. This is because prior work [32] has shown that the single-bit flip fault model is accurate for representing SDC-causing errors in programs. Moreover, the single bit-flip fault model is often as accurate as higher-fidelity models that represent transient faults at the Register Transfer Logic (RTL)-level [33].

We do not consider faults in memory elements such as caches, as they can be protected by error-correcting codes (ECC). Therefore, we exclude faults in weights and biases of the layers of the LLM, as they are stored in memory. Additionally, we do not consider faults in the instruction encoding or in the control logic of the processor, and we also assume that these faults do not modify the control flow graph of the LLMs. We make these assumptions to ensure that our injected faults lead to silent data corruptions (SDCs) instead of software crashes, which are easier to detect. Our assumptions are in line with most prior work [16, 34–36].

Finally, we primarily consider faults during LLM inference rather than during training. This is because training is a one-time process (for most LLMs), while inference is repeatedly invoked during an LLM's deployment. Furthermore, because LLMs consume significant computational and memory resources, traditional techniques to protect from hardware faults, such as Triple-Modular Redundancy (TMR), are too expensive in practice to protected them.

III. Fault Injection Methodology

The primary challenge we faced while performing FI in LLMs is scalability. LLMs like CodeBert [3] have 124.6 Million parameters and execute approximately 30 Billion CPU instructions at runtime to produce output. Therefore, we had to make modifications to the LLTFI tool to make it work with LLMs as it was not originally designed for this purpose.

1. FI Runtime: We added a supplementary FI runtime that is essentially a stripped-down and highly-optimized version of LLTFI's original FI runtime. We then statically linked this runtime with the LLM's executable. Our supplementary FI runtime reduced FI overhead by about 25% (on average), thus making LLTFI more scalable.

2. Fault propagation tracing: LLTFI supports ML fault propagation tracing that allows users to ascertain the origin, propagation, or suppression of the injected fault. However, LLTFI's original ML fault propagation tracing is not scalable to LLMs. Therefore, we replaced the element-wise output tensor comparison, i.e., element-wise comparison of each layer's output with and without FI, with hash-based comparison, where we just compare the hashes of the layer's output. While the hash-based comparison is scalable, it does not reveal the absolute amount of difference between the layer's outputs after FI. However, this optimization prevents us from ascertaining whether the fault corrupts just one element in the layer's output or multiple elements.

Figure 2 shows how we perform FIs using LLTFI in LLMs. The steps are as follows:

- **Step 1:** We first download the LLM from the Hugging Face model hub [37]. Hugging Face's Model hub contains a large collection of open-source pre-trained LLMs, along with their tokenizer and vocabulary files.
- **Step 2 and 3:** We also download the tokenizer and vocabulary files of the LLM that we use to convert human-readable text inputs to tokens in the TensorProto format [38]. TensorProto defines a serialized format for representing multi-dimensional arrays of numerical data that we used for passing data to and from an LLM.
- **Step 4 and 5:** We then convert the downloaded LLM into the ONNX format, followed by lowering down to the LLVM IR, using the ONNX-MLIR tool, similar to the default working of LLTFI.
- **Step 6 and 7:** Subsequently, we feed the LLVM IR file and inputs of the LLM in TensorProto format to LLTFI, which executes various FI campaigns. The resulting output of each FI campaign is then transformed - through our custom scripts - into a human-readable format utilizing the tokenizer and vocabulary files of the LLM and thereafter compared with the ground truth.

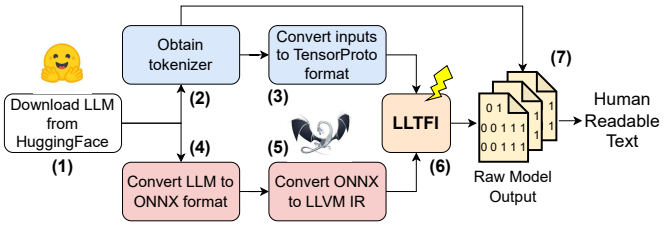


Fig. 2: Workflow of fault injection in LLMs using LLTFI.

IV. Evaluation Methodology

In this section, we describe the research questions we asked in this work, followed by the experimental setup.

A. Research Questions

We asked the following research questions (RQs) to evaluate the resilience of LLMs under transient faults.

- **RQ1:** How do SDCs manifest in the LLMs?
- **RQ2:** How does the SDC probability vary across the different layers of the LLMs and the faulty bit position?
- **RQ3:** How do the SDC rates of LLMs vary with different pre-training objectives, fine-tuning objectives, and the number of encoder/decoder blocks?

B. Experimental setup

ML applications. For RQ1 and RQ2, we chose five benchmarks: PubMedBert [39], CodeBert [3], T5 [20], GPT2 [23], and Roberta [22], as shown in Table I. These benchmarks vary in their architecture: three of them are Encoder-only, one Decoder-only, and one with Encoder-Decoder architecture. We chose these benchmarks based on their popularity in their respective domains i.e., how many times they were downloaded in the last month at the time of writing. PubMedBert (100K) for Medical-related fill-mask and question answers, CodeBert (200K) for code completion in Java, T5 (6 Million) for language translation, Roberta (11 Million) for sentiment analysis, and GPT2 (21 Million) for text generation.

For RQ3, we used fifteen benchmarks from Alajrami et al.'s work [40] on evaluating the effect of different pre-training objectives on LLM's learning capabilities. All of these benchmarks are variants of *Bert-base*, *Bert-small*, and *Bert-medium*, and are pre-trained with five different objectives each, namely (1) Masked Language Modeling [21], (2) Manipulated word detection [41], (3) Masked first character prediction [41], (4) Masked ASCII code summation [40], and (5) Masked Random token classification [40]. We explain these objectives in Section V-D.

For this work, we further fine-tuned these benchmarks on two tasks: fill-mask and question-answers, thereby resulting in a total of 30 distinct benchmarks.

Datasets. We used the pre-trained, publicly-available models of PubMedBert and CodeBert, trained with the

PubMed [42] and CodeSearchNet [43] datasets, respectively. T5 was first pre-trained on the C4 dataset (using an unsupervised denoising objective) and then trained again on CoLa [44], MNLI [45], and 12 other datasets (using supervised training objectives). Roberta was pre-trained on Stories [46] and four other datasets. GPT2 was primarily trained on the WebText dataset [47], which consists of about 40GB of text.

	Benchmark Name	Architecture	Task type	# of Parameters
RQ1 & RQ2	PubMedBert [39]	Encoder-Only	Fill-mask	109.5M
	CodeBert [3]	Encoder-Only	Fill-mask	124.6M
	T5 [20]	Encoder-Decoder	Translation	222.9M
	Roberta [22]	Encoder-Only	Sentiment Analysis	124.6M
	GPT2 [23]	Decoder-Only	Text Completion	124.4M
RQ3	Bert-base*	Encoder-Only	Fill-Mask, Question Answer	108.3M
	Bert-medium*			51.5M
	Bert-small*			38.9M

TABLE I: Benchmarks used for resilience evaluation of LLMs and the RQs in which they are used. Benchmarks marked with * are compiled with five different pre-training objectives.

Benchmarks used in RQ3 are primarily pretrained on BookCorpus [48] and English Wikipedia. We further fine-tuned these models using the Eli5 [49] and SQuAD [50] datasets for the fill-mask and question-answer tasks, respectively.

Fault Injection (FI). For each of the five benchmarks used for RQ1 and RQ2, we used ten distinct inputs from their training datasets, while for each of the 30 benchmarks used in RQ3, we used five distinct inputs in order to keep our FI experiments computationally tractable. For each input, we ran 1000 FI experiments. For the T5 benchmark, we performed FI separately for the encoder and decoder in order to explore the resilience of each component. Therefore, throughout this work, we use the term "T5-encoder" and "T5-decoder" to denote T5 with FI in the encoder and decoder, respectively. This resulted in a total of $210,000$ FI campaigns - $60,000$ for RQ1 and RQ2, and the remaining for RQ3 - for all our experiments, *which took approximately 336 Hours of CPU time*. Note that the model weights of all of our benchmarks are static, i.e., in the absence of FI, the input-output mapping of our benchmarks is constant across the FI trials.

In this work, we injected only single bit-flip faults as this is the de-facto fault model used by prior work [16, 25, 36] (Section II-C). Moreover, we injected only a single fault in each FI trial as transient hardware faults are rare events relative to the execution of LLMs, and it is multiple transient faults in a single LLM execution. Additionally, to avoid application crashes, we introduced errors only in the operands and outcomes of floating-point arithmetic and logical instructions such as FADD, FMUL, and FCMP (for the PubMedBert benchmark, these instructions constitute ~15% of the total CPU instructions). This FI methodology ensured that the control flow of the LLM remained uncorrupted - this is in line with prior work [16] and our fault model (Section II-C). To inject a single bit-flip fault in an LLM, we first randomly select a layer, followed by randomly selecting a floating-point arithmetic instruction at runtime to inject the fault into, and finally, we randomly select a bit in the operands or results of the chosen instruction to inject the bit-flip fault.

Metrics. To evaluate LLMs' resilience, we used two metrics: (1) SDC rate and (2) cosine similarity. An SDC is a misprediction (the output of the LLM differs from the correct one), and the SDC rate refers to the fraction of injected faults that result in a misprediction. Note that all our benchmarks have static weights, i.e., for a given input, the output of the LLM is fixed, regardless of the number of trails. Therefore, when the output of the LLM differs from the correct one (due to the injected fault), we classify it as an SDC. The SDC rate has been widely used as a metric by most other FI studies [16, 25, 35, 36] (though they are not on LLMs).

On the other hand, cosine similarity is a popular metric in the NLP community [51, 52] to compare the similarity between document vectors or word embeddings. While the SDC rate measures how frequently the output differs from the correct prediction, cosine similarity measures how distant the corrupted output is from the correct one, due to the fault.

To calculate cosine similarity, we converted the text output of the LLM into vector embeddings using the FastText [53] embeddings. For T5 (translation of English-to-German and English-to-French), we used FastText's German and French variants, publicly available in FlairNLP [54]. For the CodeBert benchmark, since the output of this benchmark is Java code, we could not use the cosine similarity metric due to the lack of open-source Java word embeddings. Similarly, for Roberta, the output is just 'Positive' or 'Negative', and hence we could not use the cosine similarity metric for this benchmark.

For both the SDC rate and the cosine similarity, we calculate values using 95% confidence intervals. Since SDCs are rare events, for the SDC rate, we used the confidence intervals proposed by Leveugle et al. [55], which is in line with prior work on FI [25, 56]. For cosine similarity, we calculate the non-parametric 95% confidence intervals.

Limitations of our metrics: SDC rate, i.e., strict word-to-word comparison of the corrupted and the correct output, might not capture the polymorphic nature of natural languages, where we can express the same concept using different words. While cosine similarity of word embeddings can capture the polymorphic nature of natural languages upto a large extent, it, too, can not fully capture the contextual nuances and ambiguities present in natural language. For example, the cosine similarity of *"This chair is white and the table is black."* and *"This chair is white and the table is white too"* (example taken from our GPT2 benchmark) is over 90%, indicating a near-perfect match when in reality, these two sentences have completely different meanings.

To address these limitations, in Section V-B, we present a qualitative categorization of SDCs, based on their syntactical and semantical differences relative to the correct output.

Setup. We ran all our experiments on an 64-bit, AMD

Benchmark Name	Total SDCs	Syntactically-incorrect SDCs	Semantically-incorrect SDCs	Semantically-correct SDCs
PubMedBert	77	46	14	17
CodeBert	38	35	3	0
T5-encoder	119	42	29	48
T5-decoder	192	154	24	14
GPT2	103	45	49	9
Roberta	23	0	23	0

TABLE II: A qualitative categorization of SDCs.

Ryzen Threadripper 3960X 24-Core Processor, with three NVIDIA RTX A5000 GPUs.

V. Results

In this section, we first present the SDC rates of our benchmarks. We then present the results based on the RQs.

A. SDC Rates

Table III shows the SDC rates and cosine similarity metrics for all of our benchmarks. Overall, the average SDC rate across our benchmarks varied from 0.002 (in the case of Roberta) to 0.019 (for the T5 decoder), with the average being 0.009. Regarding cosine similarity, we observed that for PubMedBert and GPT2, the average cosine similarity is 0.3, indicating that the corrupted output differs significantly from the correct one. On the other hand, for T5-encoder and T5-decoder, the average cosine similarity is 0.8, indicating that the corrupted output is semantically very similar to the correct output (see Section V-B for more details).

We make two observations from the results.

Observation 1: Large variance of SDC rate for different inputs of the same model: For PubMedBert, the SDC rate varies from 0.002 to 0.015 (approx. seven times) for different inputs. Similarly, for T5-encoder and T5-decoder, the SDC rate varies from 0.005 to 0.026 and from 0.016 to 0.029, respectively. Thus, the SDC rate varies widely across inputs.

Observation 2: Large variance of SDC rate for different benchmarks with the same architecture: Even though PubMedBert, Roberta, and CodeBert, have encoder-only LLM architecture with the same number of transformer blocks, their SDC rates vary from 0.008 (for PubMedBert) to 0.002 (for Roberta). This observation indicates that, apart from the LLM architecture, other factors like training data set, training objectives, fine-tuning task, etc., also play an essential role in determining the SDC rates, thus prompting us to ask RQ3.

B. RQ1: Qualitative categorization of SDCs in LLMs

In this RQ, we delineate how the SDCs manifest in different LLMs by providing examples of SDCs, along with a qualitative categorization of SDCs, depending on whether they are syntactically incorrect, semantically incorrect, or semantically correct with respect to the original output of the LLM. The objective of this qualitative categorization is to augment our quantitative analysis in Table III.

For this analysis, we manually analyzed all the corrupted outputs of the LLMs (552 SDCs, across all benchmarks and their inputs) and categorized them as syntactically incorrect, semantically incorrect, or semantically correct. We consider the output of the LLM to be syntactically incorrect if the corrupted output does not form grammatically correct text in the target language. We consider the output to be semantically incorrect if the corrupted output is syntactically correct but its meaning differs from that of the original text. Finally, we consider the output to be semantically correct if it is syntactically correct, *and* its meaning is similar to that of the original text, based on our subjective evaluation.

Table II shows the categorization of SDCs across all benchmarks. We explain the results for each benchmark below.

1) PubMedBert: For the PubMedBert benchmark, among the 77 SDCs observed across all inputs (out of 10,000 FI campaigns), 46 are syntactically incorrect, 14 are semantically incorrect, and the remaining 17 are semantically correct. The following is an example of the semantically incorrect output (mistakes are underlined in the examples below):

Effect of transient faults on PubMedBert (fill-mask)
Input: The hereditary [MASK] protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.
Correct Output: The hereditary plp protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.
Corrupted Output: The hereditary myb protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.

Here, the model predicts the *myb* protein instead of the *plp* protein, which is incorrect in this context. Similarly, we observed cases where the model, under the effect of the fault, predicted the wrong disease and diagnosis, which are semantically different from the correct output. However, for syntactically incorrect outputs, the model ends up predicting out-of-context characters, including non-English language characters, punctuation marks, numbers, etc. The following is an example of the syntactically incorrect output: "Main symptom of common flu is ##合.", where the model outputs out-of-context Chinese (Hanzi) characters.

2) T5-decoder: For T5-decoder, among the 192 observed SDCs, 154 are syntactically incorrect, 24 are semantically incorrect, and 14 are semantically correct. The following shows an example of each type of SDC for this application:

Effect of transient faults on T5-decoder (translation)
Input: translate English to French: The House rose and observed a minute's silence
Correct Output: L'Assemblée se levera et observera une minute de silence
Semantically-correct Output #1: Le Parlement se levera et observera une minute de silence
Semantically-incorrect Output #2: zaharie a eu l'occasion de s'exprimer
Syntactically-incorrect Output #3: zügliche de l'Assemblée

For most semantically-correct SDCs, the model either ended up using synonyms of words of the original text (like in the example above) or just paraphrasing the correct output. In the semantically incorrect SDCs, the model outputs a coherent text in the target language, but its meaning differs significantly from the correct text. For instance, in the example above, the model outputs a "zaharie a eu l'occasion de s'exprimer", which translates to "Zaharie had the opportunity

Benchmark Name	SDC Rate and Cosine Similarity (CS)											
	Input 1	Input 2	Input 3	Input 4	Input 5	Input 6	Input 7	Input 8	Input 9	Input 10	Average	
PubMedBert	0.012 ± 0.006	0.005 ± 0.004	0.002 ± 0.002	0.015 ± 0.007	0.004 ± 0.003	0.010 ± 0.006	0.007 ± 0.005	0.005 ± 0.004	0.006 ± 0.004	0.011 ± 0.006	0.008	**SDC**
	0.27 ± 0.05	0.26 ± 0.06	0.28 ± 0.05	0.25 ± 0.07	0.27 ± 0.06	0.29 ± 0.05	0.15 ± 0.08	0.28 ± 0.05	0.15 ± 0.07	0.22 ± 0.12	0.24	**CS**
CodeBert	0.004 ± 0.003	0.003 ± 0.003	0.004 ± 0.003	0.005 ± 0.004	0.002 ± 0.0027	0.003 ± 0.003	0.007 ± 0.005	0.003 ± 0.003	0.003 ± 0.003	0.004 ± 0.003	0.004	**SDC**
	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	**CS**
T5 encoder	0.013 ± 0.007	0.010 ± 0.006	0.009 ± 0.005	0.015 ± 0.007	0.026 ± 0.009	0.005 ± 0.004	0.005 ± 0.004	0.014 ± 0.007	0.007 ± 0.005	0.015 ± 0.007	0.012	**SDC**
	0.87 ± 0.13	0.76 ± 0.17	0.68 ± 0.25	0.80 ± 0.15	0.86 ± 0.09	0.86 ± 0.26	0.66 ± 0.30	0.85 ± 0.13	0.61 ± 0.24	0.70 ± 0.18	0.76	**CS**
T5 decoder	0.019 ± 0.008	0.029 ± 0.01	0.017 ± 0.008	0.017 ± 0.008	0.016 ± 0.007	0.018 ± 0.008	0.018 ± 0.008	0.023 ± 0.009	0.016 ± 0.007	0.019 ± 0.008	0.019	**SDC**
	0.85 ± 0.02	0.83 ± 0.02	0.85 ± 0.017	0.87 ± 0.028	0.84 ± 0.02	0.85 ± 0.018	0.85 ± 0.04	0.86 ± 0.017	0.84 ± 0.04	0.85 ± 0.06	0.85	**CS**
GPT2	0.015 ± 0.007	0.019 ± 0.008	0.007 ± 0.005	0.019 ± 0.008	0.007 ± 0.005	0.007 ± 0.005	0.005 ± 0.004	0.006 ± 0.004	0.006 ± 0.005	0.005 ± 0.004	0.010	**SDC**
	0.59 ± 0.18	0.30 ± 0.06	0.19 ± 0.08	0.29 ± 0.04	0.44 ± 0.01	0.44 ± 0.01	0.50 ± 0.15	0.21 ± 0.01	0.27 ± 0.25	0.35 ± 0.04	0.36	**CS**
Roberta	0.002 ± 0.0027	0.001 ± 0.002	0.003 ± 0.003	0.002 ± 0.0027	0.004 ± 0.003	0.002 ± 0.0027	0.006 ± 0.004	0.000 ± 0	0.001 ± 0.002	0.002 ± 0.0027	0.002	**SDC**
	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	**CS**

TABLE III: SDC rates and cosine similarities of all our benchmarks. Higher SDC rate implies lower resilience. Higher cosine similarity value indicates that the corrupted output is closer to the correct one.

to express himself " in English, which differs from the original text (*"The house rose and observed a minute's silence"*).

For syntactically incorrect SDCs, along with illegible outputs, we also observed many cases where some part of the output is in a different language than the rest of the text. For instance, in the above example, the model outputs "zügliche de l'Assemblée" where "zügliche" is a German word (for 'prompt') while the rest of the sentence is in French. In another interesting example, the model outputs "Ich aparitie im Namen des Europäischen Parlaments" as the German translation of "I would like, on behalf of the European Parliament,". While the rest of the text is in German, the word "aparitie" is Romanian and means "to appear or appearance" - this word fits correctly in the context of this text.

3) T5-encoder: For T5-encoder, among the 119 total SDCs we observed, 42 are syntactically incorrect, 29 are semantically incorrect, and the remaining 48, are semantically correct. Thus, unlike other benchmarks (except Roberta), SDCs in the T5 encoder are more likely to result in syntactically correct outputs. The following are examples of semantically correct and incorrect outputs in this application.

Notice that in the semantically-incorrect output shown in the above example, the model uses the word "gratitude" instead of "sympathy", which drastically changes the meaning of the original text. We have observed similar cases where the model's output is very close to the correct text but is still semantically incorrect.

Effect of transient faults on T5-encoder (translation)
Input: translate English to French: I should like, on behalf of the European Parliament, to express our sympathy to the parents and families of the victims.
Correct Output: Au nom du Parlement européen, je voudrais exprimer notre sympathie aux parents et aux familles des victimes.
Semantically-correct Output #1: Au nom du Parlement européen, je voudrais exprimer notre sympathie aux parents des victimes.
Semantically-incorrect Output #2: Au nom du Parlement européen, je voudrais exprimer notre gratitude aux parents et aux familles des victimes.

For example, the model outputs "Ich erkläre die am Freitag, dem 12. Dezember 2000, unterbrochene Sitzungsperiode des Europäischen Parlaments" as the German translation of "I declare resumed the session of the European Parliament adjourned on Friday, 15 December 2000." Notice how the model subtly changes the date in the original prompt due to the fault, making it difficult to catch.

4) Roberta: The output of Roberta is either "Positive" or "Negative," depending on the sentiment of the input text. We did not observe any syntactically-incorrect output. Therefore, all the SDCs are semantically incorrect by definition. The following is an example of the semantically-incorrect output:

Effect of transient faults on Roberta (sentiment analysis)
Input: I really like the new design of your website!
Correct Output: Positive
Semantically-Incorrect Output: Negative

5) CodeBert: For CodeBert (Java code completion), among 38 SDCs observed, we found 35 of them to be syntactically incorrect, i.e., the resulting Java code is syntactically wrong. Only three SDCs are semantically incorrect, and none of them is semantically correct. The following shows an example of a semantically incorrect SDC in this application.

Effect of transient faults on Codebert (fill-mask)
Input:
protected Iterator $\langle Map.Entry\langle K, V\rangle\rangle$
$createEntrySetIterator()$ {
if ($size()$ **[MASK]** 0) {
$return\ EmptyIterator.INSTANCE;$ }
$return\ new\ EntrySetIterator\langle K, V\rangle(this);$
}
Correct Prediction: $==$
Semantically-incorrect Output: $>=$

In this example, the model predicts a wrong binary operator ($>=$ instead of $==$), which is semantically incorrect as it could modify the functionality of the program by returning an empty iterator even if the `Map` has a non-zero size. The low SDC rate (0.004, on average) for this benchmark and the high number of syntactically incorrect SDCs imply that CodeBert is quite resilient to transient faults: even in the case of SDCs, the corrupted output would be easier to detect.

6) GPT2: Unlike other benchmarks, which are used for text translation (T5), fill-mask (PubMedBert and CodeBert), or sentiment analysis (Roberta), GPT2 is used for text generation, and therefore, there is no objective ground truth for GPT2. Therefore, to categorize SDCs, we compare the SDC, syntactically and semantically, with the model's output in the absence of FI. Additionally, we truncated the output of GPT2 to one sentence, for readability.

Among the 103 observed SDCs, we found 45 of them to be syntactically incorrect, i.e., they do not form a coherent English sentence, 49 are semantically incorrect, and nine are

semantically correct (relative to the model's output in the absence of FI). The following example shows two of the semantically incorrect outputs we observed.

Effect of transient faults on GPT2 (text completion)
Input: This chair is white and the table is
Correct Output: This chair is white and the table is black
Semantically-incorrect Output #1: This chair is white and the table is white too.
Semantically-incorrect Output #2: This chair is white and the table is lawmakers and the media who claim you are the boss of Rep.

C. *RQ2: Distribution of SDCs across layers of LLMs and faulty bit position*

Figure 3 shows the distribution of SDCs across LLM's layers and bit positions across our benchmarks. The subfigure for each benchmark in Figure 3 shows two stacked bar plots: one highlighting the distribution of SDCs along the layers of the LLM, and the other highlighting the distribution of SDCs in different bit locations of the fault.

For all our benchmarks, the layers of LLMs (and their corresponding transformer blocks) are equally sensitive to transient hardware faults. Prior work [11, 57] on FI in Convolutional Neural Networks (CNNs) has found the first and last layers of DNN to be more sensitive to transient faults. However, we do not observe a similar trend for LLMs.

Regarding the distribution of SDCs along different bit positions, in general, we found the higher-order bits to be significantly more sensitive to transient faults (this was the case for CNNs as well [11, 25]). Since all our benchmarks use Float32 datatype, higher-order bits (24 to 31) correspond to the exponent of the floating-point value, and therefore, a bit-flip in higher-order bits alters the exponent values.

Moreover, we used LLTFI's ML fault propagation tracing to understand why some transient faults get suppressed while others result in an SDC. We observe that only the 0-to-1 bitflips in higher-order bits, which increase the value of the exponent, result in an SDC. However, none of the 1-to-0 bitflips in the exponent resulted in an SDC. Additionally, we observe that faults in lower-order bits often get suppressed, either by multiplication with zero and near-zero values or by the normalization layers of the transformer blocks.

Finally, from Figure 3, we did not observe any discernible relationship between the category of SDC (syntactically incorrect, semantically incorrect, and semantically correct) and the bit location of the injected fault, across all benchmarks, except T5. For T5-encoder and T5-decoder, we observe that transient faults in 26 - 29 bits are more likely to result in a syntactically-correct SDC.

D. *RQ3: Variation of SDC rates with different pre-training objectives, fine-tuning objective, and the number of encoder/decoder blocks*

Table IV shows the variation of SDC rates with different pre-training, fine-tuning objectives, and the size of LLMs (number of encoder/decoder blocks). For this RQ, we used three different sizes of the Bert LLM: *Bert-Small* with four encoder blocks, *Bert-medium* with eight encoder blocks, and *Bert-Base* with 12 encoder blocks. We further used two fine-tuning objectives: Fill-mask (fill-in-the-blanks) and Question Answer, and pre-trained the models with the following five different pre-training objectives each:

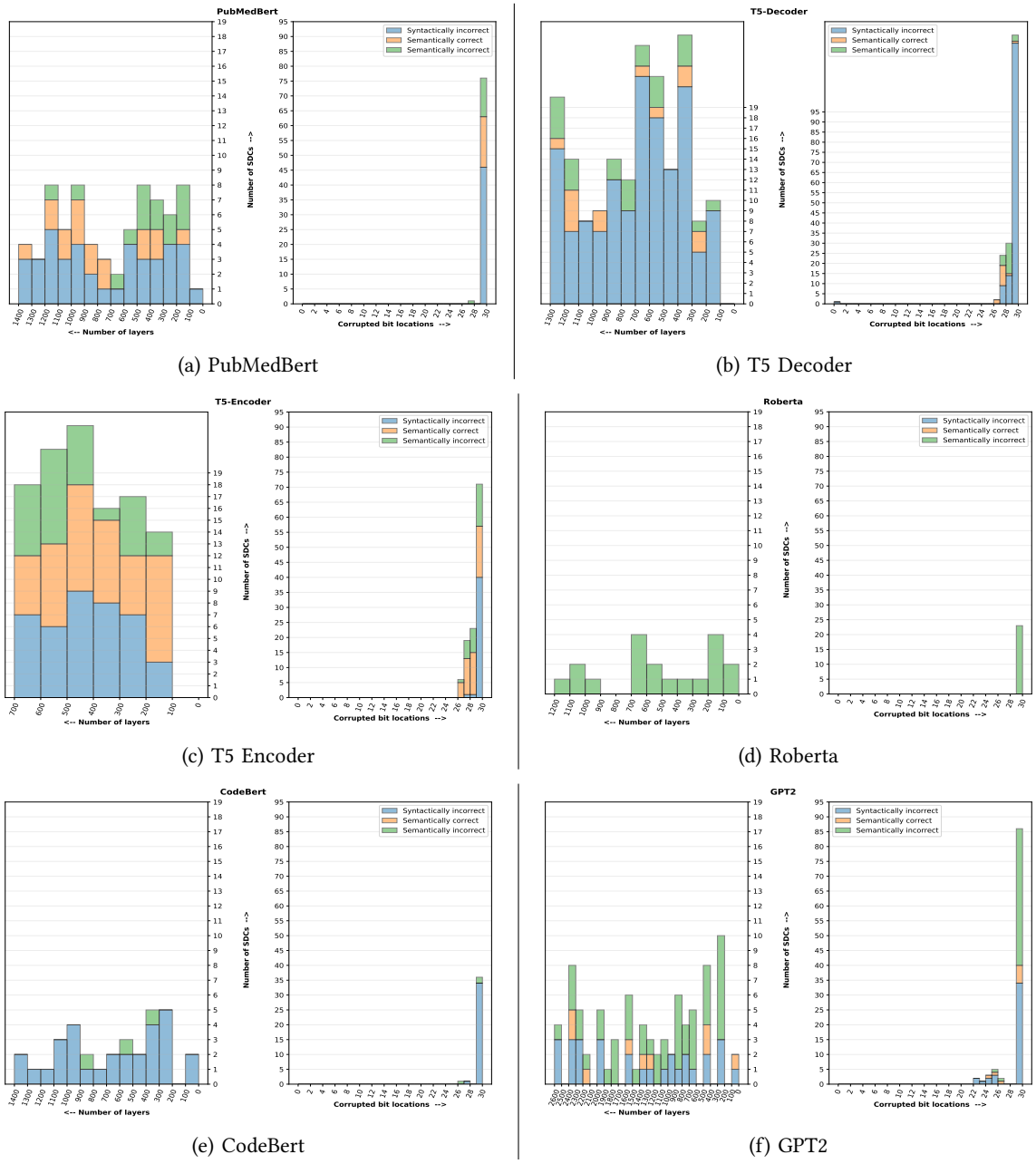
1) Masked Language Modeling (MLM) [21], which randomly chooses tokens from the input and replaces them either with the *[MASK]* token or with a random token.
2) Manipulated word detection (S+R) [41], which randomly chooses tokens from the input and either replaces them with random tokens or with shuffled tokens from the same input.
3) Masked first character prediction (FC) [41], where the model is trained to predict just the first character of the masked token.
4) Masked ASCII code summation (ASCII) [40], where the model is trained to predict the summation of the ASCII codes of the masked token.
5) Masked Random token classification (Random) [40], where tokens from the input sequence are randomly masked into five different classes and the model has to predict the class of the masked token. This objective prevents the model from learning any meaningful linguistic information.

Effect of model size on SDC rates: For the fill-mask fine-tuning objective, we observe an increase in the SDC rate with the model size (except for the *Bert-Random* pre-training objective): with *Bert - Small* the average SDC rate was 0.0026 across all inputs, for *Bert - Medium* the average SDC rate was 0.004, and for *Bert - Base* the average SDC rate was 0.007. However, for the Question-Answer fine-tuning objective, we do not observe any relation between the SDC rate and model size. Understanding the reasons is a subject for future work.

Effect of fine-tuning objectives on SDC rates: For *Bert-Small*, we observe a significant increase ($8\times$) in the SDC rate for the question-answer fine-tuning objective compared to the mask-fill objective. However, for *Bert-Medium* and *Bert-Base*, the difference in the SDC rates between the two fine-tuning objectives is small ($1.7\times$ and $1.25\times$ for *Bert-Medium* and *Bert-Base*, respectively).

We investigate why the fill-mask fine-tuning objective has a lower SDC rate than the question-answer objective. In fill masks, we found that a large portion of faults (in higher-order bits) propagates to the final layer but gets suppressed by the model's tokenizer. Thus, the model outputs N tokens, where N is the vocabulary size (28K), V, multiplied by the number of tokens, T, in the input text. The tokenizer, however, masks $V \times (T-1)$ tokens, thus masking potential SDCs.

Effect of pre-training objectives on SDC rates: We do not observe any relationship between the SDC rates and the pre-training objectives. However, for the *Bert-Random* objective, we observe that the SDC rate is the same for a given fine-tuning objective, irrespective of the model size. This is likely because in the *Bert-Random* objective the model does not learn any linguistic features of the language and thus, increasing the model size does not help the model



(a) PubMedBert (b) T5 Decoder

(c) T5 Encoder (d) Roberta

(e) CodeBert (f) GPT2

Fig. 3: Distribution of SDCs across LLM's layers and the bit position (in a 32-bit float value) in which fault is injected. Subfigure for each benchmark shows two plots: The number of SDCs (Y-axis) Vs. the layer in which the fault is injected (-X-axis) and the Number of SDCs (Y-axis) Vs. the bit position in which fault is injected (+ X-axis). Lower values are better.

learn model new linguistic features. We will investigate these effects further in future work.

VI. Related Work

Resilience of ML models under transient faults

1) Inference-time fault injection: Li et al. [11] were among the first to analyze the effects of transient hardware faults on DNNs by conducting a systematic FI study in a DNN simulator, specifically during the inference phase. Following their work, Reagen et al. presented Ares [36] introduced Ares, a fault injection tool that leverages tensor operations on the GPU to enhance the speed of fault injection at the application level. Ares, however, requires changes in the Keras inference computation graph which require changes in the Keras and Tensorflow framework. To overcome this limitation, researchers proposed PyTorchFI [58] and TensorFI [25], which are application-level FI tools developed for PyTorch and TensorFlow ML framework, respectively, and do not require changes in the ML framework itself. As pointed out by Agarwal et al. [16], application-level fault

Fine-tuning Objective ->	Mask-Fill						Question-Answer					
Pretraining Objective	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**
	Bert - Small											
Bert - MLM	0.004	0.002	0.002	0.002	0.001	**0.0022**	0.011	0.007	0.006	0.007	0.006	**0.0074**
Bert - S+R	0.0	0.003	0.002	0.003	0.002	**0.002**	0.009	0.01	0.006	0.004	0.011	**0.008**
Bert - First Char	0.001	0.004	0.004	0.004	0.004	**0.0034**	0.005	0.004	0.006	0.008	0.003	**0.0052**
Bert - ASCII	0.0	0.003	0.004	0.004	0.002	**0.003**	0.008	0.005	0.008	0.004	0.005	**0.006**
Bert - Random	0.0	0.003	0.004	0.004	0.004	**0.003**	0.008	0.010	0.009	0.011	0.004	**0.084**
	Bert - Medium											
Bert - MLM	0.0	0.008	0.008	0.003	0.004	**0.005**	0.005	0.007	0.006	0.004	0.003	**0.005**
Bert - S+R	0.003	0.005	0.001	0.005	0.003	**0.003**	0.011	0.010	0.003	0.011	0.007	**0.008**
Bert - First Char	0.0	0.004	0.006	0.005	0.007	**0.0044**	0.010	0.007	0.006	0.010	0.006	**0.008**
Bert - ASCII	0.003	0.008	0.005	0.006	0.003	**0.005**	0.004	0.004	0.007	0.007	0.015	**0.007**
Bert - Random	0.0	0.0	0.004	0.006	0.003	**0.003**	0.009	0.007	0.008	0.005	0.006	**0.007**
	Bert - Base											
Bert - MLM	0.009	0.008	0.008	0.006	0.006	**0.007**	0.006	0.005	0.013	0.010	0.009	**0.0086**
Bert - S+R	0.011	0.009	0.005	0.006	0.010	**0.008**	0.006	0.006	0.012	0.009	0.007	**0.008**
Bert - First Char	0.013	0.008	0.006	0.006	0.003	**0.007**	0.008	0.007	0.006	0.01	0.01	**0.008**
Bert - ASCII	0.008	0.008	0.003	0.014	0.003	**0.007**	0.009	0.008	0.005	0.009	0.01	**0.008**
Bert - Random	0.003	0.005	0.002	0.004	0.003	**0.0034**	0.012	0.007	0.004	0.008	0.01	**0.0082**

TABLE IV: Variation of SDC rates with pre-training objectives, fine-tuning objectives, and number of encoder/decoder blocks.

injectors like Ares, TensorFI, and PyTorchFI have limited visibility in the ML model, which causes them to exaggerate the SDC rates. To overcome this limitation, Agarwal et al. also proposed LLTFI [16], which is an IR-level Fault Injection tool for ML applications that is ML framework agnostic. In this work, we extended LLTFI to carry out FI in LLMs.

He et al.'s [59] work is perhaps the closest to our work. They present FIdelity, a framework for evaluating the resilience of ML applications under transient faults in ML accelerators. Using FIdelity, they performed FI in RNNs and Transformer layers. However, their results are not comparable to ours because of the difference in the hardware model of FIdelity and our work (ML accelerator vs. CPU). Unlike FIdelity, in this work, we carry our FI in all layers of the LLM, not just the transformer blocks. Moreover, in this work, we also evaluated the effect of model size, pre-training, and fine-tuning objectives, on the resilience of LLMs.

Liu et al.'s [60] work is one of the very few papers that evaluate error propagation due to transient hardware faults in RNNs. They found that the fault tends to spread as it propagates through the RNN layers. They also found that the execution time of RNNs significantly varies due to FI, perhaps due to the iterative nature of RNNs. Unlike Liu et al., we focus on LLMs which are structurally different from RNNs, so their results are not comparable to ours.

Robustness of LLMs Prior work has explored various dimensions of LLM's reliability, like under adversarial attacks, noise in input prompts, and out-of-distribution user inputs. Bert-Attack [13] is one of the prominent works on adversarial attacks on LLM. They use Bert [21] to generate text-based, semantically correct, adversarial attacks against fine-tuned variants of Bert. As an improvement over Bert-attack, CLARE [61] uses three perturbations - Replace, Insert, and Merge - to generate more fluent adversarial attacks against Masked Language Modeling LLMs. CLARE has a higher rate of adversarial attack success than Bert-Attack. Unlike CLARE and Bert-Attack, which does sentence-level perturbations, character-level perturbation attacks [62] use misspellings to attack LLMs. With less malicious intent than adversarial attacks, Pandia et al. [14] evaluate the effect of noise (irrelevant text) on the LLM's ability to predict masked tokens. They found noise in input prompts to alter the LLM's contextual understanding, thereby causing it to mispredict. Following this, Schuster et al. [63] found that the presence of multiple Noun Phrases in the input prompts also challenges LLM's contextual understanding and basic entity tracking. While prior work on adversarial attacks and noise injection in LLMs also evaluates the resilience of LLMs, they are different from ours due to differences in the fault model we use.

VII. Conclusion

In this work, we experimentally evaluated the resilience of LLMs under transient hardware faults. We used LLTFI - an LLVM IR-level fault injector - to inject transient faults into LLMs, and measure their SDC rates, along with the cosine similarity between the correct and the corrupted outputs. Based on extensive FI experiments, we found LLMs to be quite resilient under hardware transient faults, with the average SDC rate being 0.9% across all benchmarks. We also performed a qualitative categorization of SDCs and found that the manifestation of SDCs varies significantly with different types of LLMs. Additionally, we evaluated the impact of model size, pre-training, and fine-tuning objectives on the SDC rates and observed that for the fill-mask fine-tuning objective, the SDC rate increases with the model size. Overall, our findings indicate that the resilience of LLMs needs further investigation, before determining whether the use of LLMs in safety-critical applications is justified.

Acknowledgments

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and a gift from Intel Corporation. We thank the Digital Research Alliance of Canada for computational infrastructure support. We thank Anushree Bannadabhavi for helping us integrate LLMs with LLTFI. Finally, we thank the anonymous reviewers of ISSRE'23 for their helpful comments.

References

[1] OpenAI, "ChatGPT," https://openai.com, 2021, accessed: May 28, 2023.

[2] GitHub, "GitHub Copilot," https://copilot.github.com, 2021, accessed: May 28, 2023.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[4] Y. Qi, X. Zhao, and X. Huang, "Safety analysis in the era of large language models: A case study of stpa using chatgpt," *arXiv preprint arXiv:2304.01246*, 2023.

[5] O. Zheng, M. Abdel-Aty, D. Wang, Z. Wang, and S. Ding, "Chatgpt is on the horizon: Could a large language model be all we need for intelligent transportation?" *arXiv preprint arXiv:2303.05382*, 2023.

[6] A. Elhafsi, R. Sinha, C. Agia, E. Schmerling, I. Nesnas, and M. Pavone, "Semantic anomaly detection with large language models," *arXiv preprint arXiv:2305.11307*, 2023.

[7] "Mercedes is bringing ChatGPT into its cars," 2023, cNN Business. [Online]. Available: https://www.cnn.com/2023/06/15/business/mercedes-benz-chatgpt/index.html

[8] "GM explores using ChatGPT in vehicles," 2023, reuters. [Online]. Available: https://www.reuters.com/business/autos-transportation/gm-explores-using-chatgpt-vehicles-2023-03-10/

[9] L. M. Luza, A. Ruospo, D. Söderström, C. Cazzaniga, M. Kastriotou, E. Sanchez, A. Bosio, and L. Dilillo, "Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1867–1882, 2021.

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, 2010.

[11] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[12] Y. Liu, J. Li, and Y. Zhuang, "Instruction sdc vulnerability prediction using long short-term memory neural network," in *Advanced Data Mining and Applications: 14th International Conference, ADMA 2018, Nanjing, China, November 16–18, 2018, Proceedings 14*. Springer, 2018.

[13] L. Li, R. Ma, Q. Guo, X. Xue, and X. Qiu, "Bert-attack: Adversarial attack against bert using bert," *arXiv preprint arXiv:2004.09984*, 2020.

[14] L. Pandia and A. Ettinger, "Sorting through the noise: Testing robustness of information processing in pre-trained language models," *arXiv preprint arXiv:2109.12393*, 2021.

[15] K. M. Collins, C. Wong, J. Feng, M. Wei, and J. B. Tenenbaum, "Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks," *arXiv preprint arXiv:2205.05718*, 2022.

[16] U. K. Agarwal, A. Chan, and K. Pattabiraman, "Lltfi: Framework agnostic fault injection for machine learning applications (tools and artifact track)," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 286–296.

[17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO'04*, 2004.

[18] J. Vig, "A multiscale visualization of attention in the transformer model," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 37–42. [Online]. Available: https://www.aclweb.org/anthology/P19-3007

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[24] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[25] G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Configurable Fault Injector for TensorFlow Applications," in *Proc. of ISSREW'18*, 2018.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[27] J. Bai, F. Lu, and K. Zhang, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[28] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, and K. Kawachiya, "Compiling onnx neural network models using mlir," *arXiv preprint arXiv:2008.08272*, 2020.

[29] F. Chollet, "Keras," https://keras.io, 2015.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, and L. Antiga, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, 2019.

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, and M. Devin, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[32] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2017, pp. 97–108.

[33] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 577–589.

[34] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proc. of DSN'12*, 2012.

[35] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," in *Proceedings of QRS '15*, 2015.

[36] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[37] "Huggingface Model Hub," Date accessed: 31 July, 2023. [Online]. Available: https://huggingface.co/models

[38] "Tensor Proto Format," Date accessed: 31 July, 2023. [Online]. Available: https://onnx.ai/onnx/api/classes.html#tensorproto

[39] Y. Gu, R. Tinn, H. Cheng, M. Lucas, N. Usuyama, X. Liu, T. Naumann, J. Gao, and H. Poon, "Domain-specific language model pretraining for biomedical natural language processing," 2020.

[40] A. Alajrami and N. Aletras, "How does the pre-training objective affect what large language models learn about linguistic properties?" *arXiv preprint arXiv:2203.10415*, 2022.

[41] A. Yamaguchi, G. Chrysostomou, K. Margatina, and N. Aletras, "Frustratingly simple pretraining alternatives to masked language modeling," *arXiv preprint arXiv:2109.01819*, 2021.

[42] pubmed, "PubMed Dataset," Date accessed: 31 July, 2023. [Online]. Available: https://huggingface.co/datasets/pubmed

[43] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[44] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 625–641, 2019.

[45] A. Williams, N. Nangia, and S. R. Bowman, "The multi-genre nli corpus," 2018.

[46] T. H. Trinh and Q. V. Le, "A simple method for commonsense reasoning," *arXiv preprint arXiv:1806.02847*, 2018.

[47] V. Liu and J. R. Curran, "Web text corpus for natural language processing," in *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006, pp. 233–240.

[48] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of*

the IEEE international conference on computer vision, 2015.

[49] A. Fan, Y. Jernite, E. Perez, D. Grangier, J. Weston, and M. Auli, "ELI5: long form question answering," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 3558–3567. [Online]. Available: https://doi.org/10.18653/v1/p19-1346

[50] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," *arXiv e-prints*, p. arXiv:1606.05250, 2016.

[51] B. Li and L. Han, "Distance weighted cosine similarity measure for text classification," in *Intelligent Data Engineering and Automated Learning–IDEAL 2013: 14th International Conference, IDEAL 2013, Hefei, China, October 20-23, 2013. Proceedings 14*. Springer, 2013.

[52] A. R. Lahitani, A. E. Permanasari, and N. A. Setiawan, "Cosine similarity to determine similarity measure: Study case in online essay assessment," in *2016 4th International Conference on Cyber and IT Service Management*. IEEE, 2016.

[53] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[54] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf, "FLAIR: An easy-to-use framework for state-of-the-art NLP," in *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, 2019, pp. 54–59.

[55] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 502–506.

[56] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, 2018.

[57] K. Adam, I. I. Mohamed, and Y. Ibrahim, "A selective mitigation technique of soft errors for dnn models used in healthcare applications: Densenet201 case study," *IEEE Access*, vol. 9, pp. 65 803–65 823, 2021.

[58] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020.

[59] Y. He, P. Balaprakash, and Y. Li, "Fidelity: Efficient resilience analysis framework for deep learning accelerators," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 270–281.

[60] T. Liu, Y. Fu, Y. Zhang, and B. Shi, "A hierarchical assessment strategy on soft error propagation in deep learning controller," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021.

[61] D. Li, Y. Zhang, H. Peng, L. Chen, C. Brockett, M.-T. Sun, and B. Dolan, "Contextualized perturbation for textual adversarial attack," *arXiv preprint arXiv:2009.07502*, 2020.

[62] B. Liang, H. Li, M. Su, P. Bian, X. Li, and W. Shi, "Deep text classification can be fooled," *arXiv preprint arXiv:1704.08006*, 2017.

[63] S. Schuster and T. Linzen, "When a sentence does not introduce a discourse entity, transformer-based models still sometimes refer to it," *arXiv preprint arXiv:2205.03472*, 2022.