Received April 14, 2022, accepted May 13, 2022, date of publication May 20, 2022, date of current version May 26, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3176617

Debugging Debug Information With Neural Networks

FIORELLA ARTUSO, GIUSEPPE ANTONIO DI LUNA, AND LEONARDO QUERZONI
Department of Computer, Control, and Management Engineering Antonio Ruberti, Sapienza University of Rome, 00185 Rome, Italy

Corresponding author: Fiorella Artuso (artuso@diag.uniroma1.it)

This work was supported in part by the Sapienza Projects under Grant RM11916B75A3293D and Grant RG12117A7CE68848, and in part by the PNRM Project "SAFE."

ABSTRACT The correctness of debug information included in optimized binaries has been the subject of recent attention by the research community. Indeed, it represents a practically important problem, as most of the software running in production is produced by an optimizing compiler. Current solutions rely on invariants, human-defined rules that embed the desired behavior, whose violation may indicate the presence of a bug. Although this approach proved to be effective in discovering several bugs, it is unable to identify bugs that do not trigger invariants. In this paper, we investigate the feasibility of using Deep Neural Networks (DNNs) to discover incorrect debug information. We trained a set of different models borrowed from the NLP community in an unsupervised way on a large dataset of debug traces and tested their performance on two novel datasets that we propose. Our results are positive and show that DNNs are capable of discovering bugs in both synthetic and real datasets. More interestingly, we performed a *live analysis* of our models by using them as bug detectors in a fuzzing system. We show that they were able to report 12 unknown bugs in the latest version of the widely used LLVM toolchain, 2 of which have been confirmed.

INDEX TERMS Bugs, compilers, debug information, neural networks, software engineering.

I. INTRODUCTION

Software running in production is highly optimized to maximize its performance according to several metrics. For compiled languages, binaries are the output of a compilation process where several optimization techniques are applied. While these optimizations are critical for the performance of the produced artifacts, they may expose unwanted behaviors observable only in the optimized case (e.g., race conditions [1], use-after-free [2], and heisenbugs [3]).

Therefore, debugging the exact version of the binary running in production is key to triaging specific problems that are otherwise impossible to reproduce. Hence, it is crucial to have a complete and reliable debugging process for optimized binaries [4]. In order to debug, users need a compiler and a debugger. The compiler, when instructed to do so, produces a binary that is composed of the binary code and additional debug information, which for UNIX-like systems is usually encoded using DWARF [5]. This information is then used by the debugger during the debugging phase. It is worth noting that while debuggers are the main users of debug info, also tools consume them too, e.g. profilers.

Preserving the correctness of debug information while optimization passes are applied is an extremely complex task. To address this challenge, compilers introduced new optimization levels (e.g., -*Og* in GCC and clang) specifically targeted at providing a reasonable tradeoff between the debugging process and optimizations applied. As a matter of fact, *Og* is described as the optimization level for the standard edit-debug lifecycle in the GCC documentation.

However, it has been recently shown [6], [7] that modern optimizing toolchains[1] often provide an unsatisfactory debug experience even when using optimization levels specifically designed for debugging. This happens when the compiler generates wrong debug information or when the debugger does not handle correctly the debug information produced. Therefore, it is important to examine debug traces looking for problems that can be generated by a bug in the toolchain.

Recent works [6], [7] used a differential testing approach where optimized and unoptimized binaries are compiled from

The associate editor coordinating the review of this manuscript and approving it for publication was Xabier Larrucea.

[1] An optimizing toolchain is the union of a compiler capable of optimizing code and a debugger.

the same source. The debug traces of the two binaries are then compared using manually defined invariants. These invariants look for the inconsistency of some information to identify suspect cases. This approach can only find bugs that impact the controlled information as predicted by the invariants' designers.

Therefore, it would be of extreme interest to build a technique for the automatic detection of incorrect debug traces without the use of manually defined rules. A deterministic solution to this problem would entail defining a formal model of the optimization passes of the compiler. This is far from trivial even limiting the scope to a single optimization pass.

In this paper, we take a data-driven approach, in which we use a large dataset to learn a statistical model of correct debug traces. The advantage of this approach lies in its black-box nature: it does not need knowledge nor makes any assumption about the internal structure of the compiler and debugger. We are interested in unsupervised techniques because a labelled dataset of correct/bugged debug traces is not available, and it cannot be generated automatically.

Specifically, we use neural networks following an anomaly detection approach. We train several models in an unsupervised way on a dataset of collected debug traces. Our hypothesis is that the networks may learn the relationships contained in correct debug traces. This has been inspired by works using a similar approach to find bugs in source code [8], [9].

We test our models on two novel datasets. A synthetic dataset of bugged debug traces and another one obtained from real bugs from the LLVM repository. Our experiments confirm that our models discriminate between bogus and correct traces. Finally, we test our models in a *live analysis* to find new bugs in the widely used LLVM toolchain (composed by the clang compiler and lldb debugger).

```
1  volatile int a,  g_5108; int b;
2  short c(){
3   return g_5108;
4  }
5  int main () {
6   c();
7   b = 0;
8   for (;
9   b < 4;
10  b++) a ;
11 }
```

Snippet 1. Clang bug 51511, wrong backtrace information at line 4.

A. MOTIVATING EXAMPLE

Snippet 1 shows a bug of LLVM found with our solution. The bug is present in the latest LLVM when compiling with optimization *-Og*. When stepping over instructions, the backtrace information of the lldb debugger wrongly shows that line 3 is inside the main function (this is probably the effect of inlining). This kind of bug could mislead a developer that sees a return instruction executed right at the beginning of the `main`, inferring that the rest of the instructions in that function will not be executed.

B. CONTRIBUTIONS

We provide the following contributions:

- We are the first to consider the problem of detecting bugs in debug information using DNNs. We use transformer architectures, trained with novel unsupervised tasks, to create (i) a network that is able to identify a wrong stepping behavior on a debug trace (*SLNet*) and (ii) a second one that is able to identify an incorrect mapping between assembly instructions and source code (*MapNet*).
- We release three new datasets[2]: a large unlabelled dataset constituted by debug traces, a dataset with debug traces containing synthetic bugs, and a manually labelled dataset containing real bugs.
- We conduct an experimental evaluation of the proposed architecture on the aforementioned dataset.
- We use our best-performing networks in a novel fuzzing system. We reported 12 bugs found in the LLVM toolchain: 2 of these bugs have been confirmed and 10 are pending analysis by the LLVM developers.

II. RELATED WORK

A. COMPILER TOOLCHAINS TESTING

The problem of testing compiler toolchains can be divided into three main branches: compiler, debugger and debug information testing. While compiler testing has been widely investigated, little attention has been paid to identifying bugs in debugger software and debug information of optimized binaries.

1) COMPILER TESTING

Compilers are widely used and complex software; this complexity makes them prone to bugs. Since compilers are used to build other production software, these bugs could result in unwanted behaviors, or worse in security-related problems [10], in different applications. For these reasons, there is a heap of works focusing on finding compiler bugs [11]–[15] using compiler testing.

Compiler testing techniques [16] are mainly based on fuzzing: compilers are fed with random programs either generated from scratch [17], [18] or obtained by mutating existing ones [19], [20]. The generation from scratch uses rules defined for the grammar of the specific language, mutation-based techniques apply mutations to programs generated with the first approach. A compiler bug can either generate a crash or create a program with unwanted behavior; the first case can be easily detected. The second case is usually detected using differential testing [10], [21], which is based on comparing the results or the behavior of programs obtained from different compilers. Another approach is metamorphic testing [22], which is based on the idea of transforming the input while detecting unexpected behavioral changes.

[2]https://github.com/FiorellaArtuso/NeuroDebug-2_Dataset

Some novel approaches follow the *Big Code trend* [23] and are based on NLP techniques. In particular, [24] and [25] use LSTM architectures for generating and mutating test cases. We remark that they only produce test cases and do not use neural networks to detect bugs.

2) DEBUGGER TESTING

Regarding debugger testing, [26] tests the correctness of javascript debuggers using a differential approach and assuming that different debugger implementations should exhibit the same behavior. Operatively, they execute the same debugging actions in parallel and they compare the corresponding results: every diverging behavior identifies a possible bug. Notice that this approach is not able to catch debug information bugs, if a debug information is wrong it will result in the same wrong behavior on different debuggers.

[27] proposes a metamorphic approach for testing the debugger in the Chromium browser. They transform both the input program and the debugging actions and they detect whether this transformation causes unexpected changes.

3) DEBUG INFORMATION TESTING

There are only two works that identify debug information errors in optimized binaries [6], [7] and none of them used neural approaches. Both works are based on differential testing and tackle the problem using likely invariants. In this approach, a source code program is compiled with and without optimization obtaining an optimized binary and an unoptimized one. Both binaries are executed using a debugger and recording the execution traces resulting in an optimized trace and an unoptimized one. A likely invariant compares these traces and triggers when a specific inconsistency is detected. This inconsistency could be caused by a bug in the debug process. An example is the Line Invariant of [7]; this invariant takes the unoptimized trace and the optimized one and triggers when there is a line appearing only in the latter. This could be dead code wrongly shown as executed. We stress the invariant is likely, thus false positives could be possible; this is true for the other invariants present in [7] as well. To the best of our knowledge, no technique exists to automatically create invariants for debug information.

B. NEURAL BUG FINDING

Several papers [28] used Deep Learning to find bugs in source code across several languages. We remark that these works focus on finding a bug inside the code and they do not detect wrong debug information.

An example is the so-called variable misuse bug that mainly occurs when the developer copies pieces of code from one place to another and forgets to adapt the used variables to the new location. To solve this problem, [8] represents a program using a semantic enriched Abstract Syntax Tree (AST) and uses a Graph Neural Network (GNN) on top of it to predict which variable should be used at a specific location. Another work is [29], which uses a two-pointer attention-based LSTM to jointly predict the bug location and the repairing variable.

Some other works focus on fixing errors arising at compile time (e.g. missing brackets). The problem with this type of errors is that compilers do not always show the correct error message or the correct bug location, thus misleading the programmer. [28] solves the problem by using an attention-based sequence-to-sequence network that takes as input the sequence of tokens representing the program and produces as output both the bug location and the correct fixed version of the bugged line.

Recent works [30], [31] use the concept of pretraining and finetuning of transformer-based architectures to generate fixes for different kinds of bugs.

Another work is [9], which proposes a Language Modeling based fuzzer for the javascript engine trained on regression javascript tests. In particular, the authors splits the AST of the regression test cases into subtrees and, by performing a preorder traversal of the tree, obtain sequences of such subtrees. The goal of their modified language model is to make the network predict which subtree comes next. They generate new test cases by mutating existing regression test cases by using the obtained language model. With this approach, the authors find lots of bugs and 3 CVE.

III. DEBUG TRACE, PROBLEM DEFINITION AND OVERVIEW

Given a source code c, an optimizing compiler generates an optimized program opt. From this program, a debugger produces a debug trace $T(opt) : [s_0, s_1, \ldots, s_n]$ which is an ordered list of elements, each representing a *step* over a machine instruction. $T(opt)$ is built through instruction by instruction step execution: we set a breakpoint on the program entry point and repeatedly step over assembly instructions until opt exits. At each step $s \in T(opt)$, we collect the source line $line(s)$ that the debugger shows in program c when executing s. This is the high-level source line of code in c that the debugger believes to be executed with step s. Moreover, we also collect the assembly instruction $asm(s)$ that the CPU executes in s. A c program is constituted by several functions $\{f_1, \ldots, f_m\}$; we divide an execution trace T into several traces $T_{f_1}, \ldots, T_{f_m}$, where T_{f_j} represents the execution of function f_j in trace T.

A. PRELIMINARY DEFINITIONS

Before moving on to the problem definition, let us provide some necessary elements.

1) SEQUENCE OF SOURCE LINES

Each function trace T_f defines a sequence of executed source code lines. Formally, $L(T_f) : [l_1, l_2, \ldots, l_k]$ is the *sequence of source code lines* present in trace T_f in the order of their appearance (a line l_j appears when a step $s_j \in T_f$ has $l_j = line(s_j)$); we remove loops by deleting consecutive sequences of repeated lines.

2) MAPPING ASSEMBLY AND SOURCE CODE

For each line l in c, we have associated a sequence of steps, each of which is a step over one of the assembly instructions used to execute line l. Thus, from T_f we extract a set $A(T_f)$ of *mapping pairs* $< [a_0, \ldots, a_m], l >$ where $[a_0, \ldots, a_m]$ is the sequence of all assembly instructions that the debugger maps to line l in trace T_f.

3) OPTIMIZATION LEVELS

Modern compilers provide several optimization levels. In this paper, we focus on *Og* that is a level created to provide a good debug experience, i.e. facilitate the debugging process by embedding accurate and complete debugging information in the compiled software, while creating optimized binary code. *Og* is supported by the two main C compilers available today: GCC and clang. We consider *Og* as it was shown to be the optimization level more prone to bugs [7].

B. PROBLEM DEFINITION

A pair compiler/debugger can be seen as a toolchain function F that maps each source code c to a debug trace $T = F(c)$. The set *Traces* of all possible traces generated by F for all valid programs[3] can be partitioned into the set of correct traces *NoBug* and the one of bugged traces *Bug*. We now define the *Correct Debug Detection* (CDD) Problem:

Correct Debug Detection Problem: Given a trace $T \in Traces$ a CDD gives as output 0 if T is a correct debug trace ($T \in NoBug$) and 1 if T is a bugged debug trace ($T \in Bug$).

It is worth noticing that a perfect toolchain F should never generate a bugged trace. We remark that our final goal is to find bugs in the toolchain, and so to find instances of programs that generate wrong debug traces under F. We are not concerned about the correctness of programs use to test F.

C. ASSUMPTIONS AND SETTING

We simplify the problem by considering only source code generated by csmith [17]. Csmith is a code generator used in many works on compiler correctness [32]–[35]. Csmith generates a random C program that is valid and free of undefined behaviors. The absence of undefined behaviors guarantees a well-defined semantic, and thus avoids the generation of meaningless debug traces. We will show that using Csmith generated programs is enough to find novel bugs in the LLVM toolchain.

D. SOLUTION OVERVIEW

All our approaches are based on training unsupervised models on a set of debug traces obtained by programs generated by Csmith. We use two models, one that learns the relationships between the source lines executed in a trace, and the other that learns the mapping between assembly instructions and C code.

[3]A program is valid if can be compiled by our toolchain and is free of undefined behaviors.

1) CONSISTENCY ON SOURCE LINES

A trace T is associated with a sequence of executed source lines $L(T)$. On this sequence we train a variation of a masked language model using a transformer architecture, we call such network *Source Lines Network* (SLNet). Our hypothesis is that a model trained on this task can identify odd stepping behaviors in debug traces (as in Snippet 1). As a matter of fact, [7] has shown that this is a frequent category of bugs, and it is particularly nefarious since a developer debugging a software could see an execution trace that is not consistent with the real execution flow, making it hard to understand what is really happening.

2) CONSISTENCY OF THE MAPPING ASSEMBLY-LINES

A trace T can be seen as a sequence of pairs, each of them mapping a sequence of assembly instructions $A(l)$ to a source line $l \in L(T)$. On these pairs, we train a transformer architecture on the task of identifying the correct mapping between assembly instructions and a source line. We use the term *Mapping Network* (MapNet) to indicate this architecture. An example of a bug found by MapNet is in Snippet 2. When stepping on line 8, lldb shows a shift assembly instruction associated with this line (`shll cl, edx`), this shift instruction should be associated with line 6 instead. We remark that no existent approach is able to find this kind of bugs.

```
1 int a, b;
2 int *g_3377 = &b;
3 int main() {
4   short c;
5   char l_3718[7][4];
6   c = 3 > 7 >> a ? 0 : 3 << a;
7   l_3718[6][3] = c;
8   (*g_3377) = l_3718[6][3];
9 }
```

Snippet 2. Clang bug 51507, wrong assembly mapped to line 8.

IV. ARCHITECTURES DETAILS AND UNSUPERVISED TRAINING TASKS

We solve the CDD problem with two architectures, the Longformer [36] used for SLNet and BERT [37] used for MapNet. Both architectures are encoder-only transformers composed of a stack of N identical layers, where each layer is composed of a multi-head self-attention mechanism and a fully connected feed-forward network.

A. SOURCE LINES NETWORK: SLNET

The SLNet is based on the Longformer architecture which has modified attention that scales linearly with sequence length, thus making it more suitable for processing long sequences of source code lines. In particular, this modified attention consists of a windowed self-attention that reduces time and space complexity by focusing only on local context.

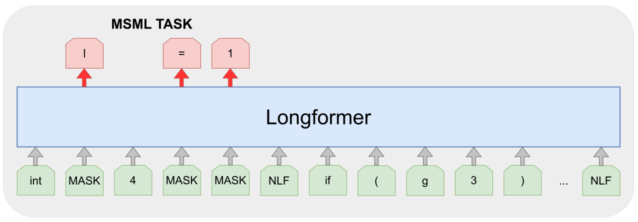

FIGURE 1. SLNet training on the masked source language modeling (MSLM) task.

1) TRAINING

SLNet is trained on the Masked Source Language Modeling Task (MSLM) which is a variation of the traditional Masked Language Modeling (MLM). The objective of MLM is to hide a certain percentage of tokens in a sentence and then teach the network to reconstruct the original tokens based on the surrounding context. We adapt this traditional NLP task to debug traces. We do so by hiding a certain percentage m_l of tokens inside one single random line l_j and teaching the network to guess them. Training is done by minimizing the reconstruction error (i.e. cross-entropy loss) of masked tokens. The training process is shown in Figure 1.

We use this variation of the original task since we are interested in detecting wrong source line stepping; this behavior manifests in real bugs as a source line that is out of context. By training the network to reconstruct a single source line in a correct trace, we expect that the reconstruction loss of a misplaced source line will be high; the network should not correctly predict a line using a wrong context. For this reason, we use higher values for m_l (≥ 0.6) than the ones used in the MLM task. Notice that even with values of masking near or equal to 1.0 is still possible to do meaningful predictions. Consider the sequence `int a=0; int b=0; int c=a+b;` and m_l is 1.0, the masked result is `int a=0; MASK MASK MASK MASK, int c=a+b;`. The model can infer that the line is initializing b, even if it is entirely masked. Note that during the training we do not use labels that identify a trace as bugged or not. Our approach is therefore unsupervised as the analogous task in Bert [37].

2) INFERENCE

We solve CDD using SLNet to compute a score for each function. For each sample, we iteratively mask m_l of the tokens inside each source line and we compute the average reconstruction error. Finally, we take the max of these values to obtain a score for each function. Figure 2 shows an example of inference with m_l equal to 0.6 on a bugged function trace composed of three lines. In *Input* 1 three tokens of the first line are masked, while *Input* 2 and *Input* 3 have one and two tokens masked in the second and third line respectively. The network is fed with all of these inputs, one at a time. For each input, we compute the averages of the losses of masked tokens which are 0.0101, 9.4 and 6.6 and we take the maximum as the score. The maximum is 9.4 and represents the score assigned to the function in question. Notice that 9.4 corresponds to the second line, which is obviously misplaced since a return instruction cannot appear in the middle of a function trace.

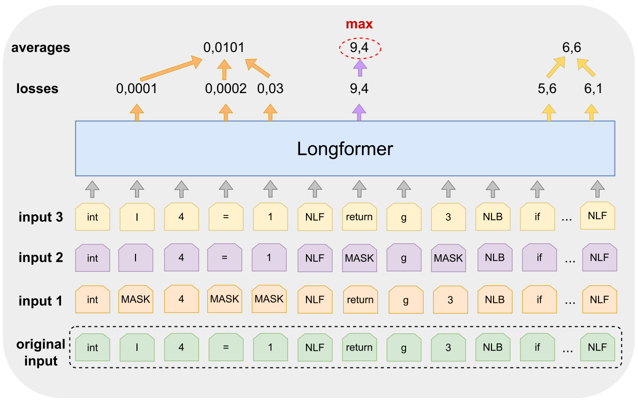

FIGURE 2. SLNet inference.

The computed score can be either used to pick the top k scored function as bug candidates or compared with a certain threshold value.

B. MAPPING NETWORK: MAPNET

The MapNet is based on the BERT architecture trained in a cross-lingual fashion [38] by taking as input parallel sentences in two different languages (assembly and C source code).

1) TRAINING

MapNet is simultaneously trained on two tasks: Asm/Source Mapping Prediction (ASMP), and Masked Asm Language Modeling (MALM). Similar to the traditional Next Sentence Prediction Task (NSP), the objective of ASMP training is to have the network predict whether a given sequence of assembly instructions is correctly mapped to a certain source line. We create a dataset for this binary classification task, from the original samples mapping a source line l to assembly instructions $A(l)$. We partition all samples in half; a partition is left untouched while the other partition is used to generate incorrect mapping pairs. The incorrect pairs are created by randomly shuffling the mapped source lines. The MALM task consists in hiding a certain percentage of tokens inside the list of assembly instructions and in teaching the network to reconstruct them. The training loss is the sum of the MALM and ASMP cross-entropy losses. The MapNet training process is shown in Figure 3. As for the MSLM task, MALM and ASMP do not use labels that identify a trace as bugged or not. Therefore, we consider our approach unsupervised as the analogous tasks in Bert [37].

2) INFERENCE

We solve the CDD using MapNet to compute a score for each function. For all samples, we use the MapNet to compute the probability that each sample represents a wrong mapping,

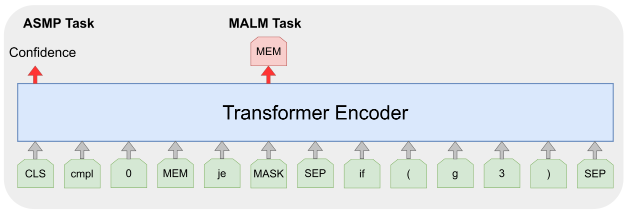

FIGURE 3. MapNet training on the Asm/source mapping prediction (ASMP), and masked Asm language modeling (MALM) tasks.

by taking the confidence value returned by the classification layer attached to the hidden state of the *CLS* token. This is also done in Bert to compute the confidence of having a pair of consecutive sentences. Then, we group all samples belonging to a function aggregating all ASMP confidence values by using the max function. This aggregated value is the score of the function. As for the SLNet, this score can be used to extract the top k samples for analysis or all the ones that are above a certain threshold.

V. DATASETS

In this section, we describe the datasets used for the experimental evaluation. We consider three distinct sets of programs: one set is used for training and validation, and the other two are used for testing. From these programs, we generate debug function traces. For MapNet, the traces are used to build datasets of samples, each being a mapping pair. For SLNet each sample is a sequence of source lines in a function trace.

A. DATASET PREPROCESSING

In this Section, we explain the preprocessing used in our datasets for SLNet and MapNet.

1) SLNET PREPROCESSING

Given a function trace T_f, we create a sample for the SLNet by first converting all hexadecimal numbers into base ten integers, and then by substituting all numbers greater than a certain threshold (we used 1,000) with a special token DEC. The lines are then tokenized using codeprep [39] removing underscore tokens. Finally, from a sequence of lines $[l_0, l_1, \ldots, l_m]$ we obtain a single string $l_0\, X_0\, l_1 \ldots X_{m-1}\, l_m$ by concatenating the lines. The token used for concatenation X_j, is either: NLF, indicating that l_{j+1} is at a higher line number than line l_j in the original program; or, NLB, indicating that l_{j+1} is at lower line number than line l_j. Practically speaking, we use NLB when the trace jumps backward (e.g., a for loop). An example of a preprocessed SLNet sample is shown in Figure 1. We removed all duplicates and all sequences of a single source line.

2) MAPNET PREPROCESSING

For the MapNet samples, we preprocess the assembly by executing a symbolication step (we substituted addresses with variable names, strings, and function names). We then substitute all the memory accesses with a special token MEM, we convert immediate operands into base ten substituting the ones above 1,000 with token DEC. We obtain a single string by concatenating all assembly instructions using whitespaces. We use the same strategy to preprocess numbers in the source line. We then preprocess source line strings as shown in the previous paragraph. Finally, the two strings are concatenated using a special token SEP. An example of a preprocessed MapNet sample is shown in Figure 3. We removed all duplicate samples. Moreover, the csmith generated programs contain many assignments of the *csmith_sink* variable and calls to the *transparent_crc* function: we performed a downsampling of samples containing these patterns in the MapNet dataset by taking 2% of them.

B. TRAINING AND VALIDATION DATASETS

We used Csmith to generate 43,921 programs and lldb to obtain 200,443 debug function traces. We split the dataset into training and validation (90%-10% split). After preprocessing, the SLNet datasets result in 81,337 traces for training and 7,195 for validation, while the MapNet datasets end with 349,153 and 25,286 mapping pairs. We analyzed the length of the samples in our datasets. For the SLNet datasets, we have that 70% of the samples contain less than 1,024 tokens, while for the MapNet we have that 99.9% of all samples contain less than 512 tokens.

C. SYNTHETIC DATASETS

These datasets were used to evaluate the performance of the model in identifying bugs. Programs inside the synthetic datasets were created as the training ones. Additionally, starting from these programs, we created both synthetic bugged traces, in which we add different types of errors, and bug-free traces, in which traces remain untouched. By observing real cases, we identified three main categories of bugs that were inserted inside the original *function traces*. Given a function trace $T_f : [s_0, s_1, \ldots, s_n]$, synthetic bugs are defined as follows:

- **swap source**: we randomly select two steps $\{s_k, s_l \mid line(s_k) \neq line(s_l))\}$ and we assign $line(s_k)$ to step s_l and $line(s_l)$ to step s_k.
- **swap assembly**: we randomly select two steps $\{s_k, s_l \mid asm(s_k) \neq asm(s_l))\}$ and we assign $asm(s_k)$ to step s_l and $asm(s_l)$ to step s_k.
- **remove step**: we randomly select a step s_j and we remove it from T_f.

We generated 3,983 programs and then we uniformly inserted one of these bugs inside one function trace for 27% of the programs. The insertion of a bug in a trace creates a single bugged sample for the SLNet dataset, while it impacts one or more samples in the MapNet dataset. Therefore, in MapNet we mark as bugged all samples deriving from a bugged trace. After duplicate removal, for the SLNet dataset, we have 6,009 non-bugged samples and 630 bugged samples (Swap Source: 338; Swap Assembly: 149; Remove Step: 143). For the MapNet dataset, we have 90,739 non-bugged samples and 10,653 bugged samples (Swap Source: 5,352; Swap Assembly: 3,093; Remove Step: 2,208).

D. REAL BUGS DATASETS

These datasets were created by using real bugs from the LLVM repository. We analyzed 42 bug reports and identified bugs that could exhibit a wrong step behavior or a wrong mapping assembly/source. For each bug, we created a program that generates the reported bugged behavior and obtained a trace using the LLVM toolchain version containing that bug. In the program, we normalized variables and function names using the same naming convention used by csmith. At the end of this process, we obtained 16 different programs. When possible, for each program we created a trace that does not contain the bug by using a patched toolchain. Note that this generates a challenging dataset since the only difference between bugged/non-bugged traces is the presence of the bug itself; both traces are derived from the same program.

We obtained a dataset for SLNet with 29 traces, 18 bugged and 11 bug-free. For MapNet we obtained 136 samples, 76 bugged and 60 bug-free.

VI. EXPERIMENTAL EVALUATION

This Section reports the results of our experimental evaluation on the synthetic and real datasets.

A. TRAINING, MODELS PARAMETERS, AND METRICS

Since our amount of training data is limited, we use smaller architectures than the ones usually adopted in NLP. We use [40] as a guide for the selection of parameters for smaller transformer models. We test medium models composed of 8 layers, 8 attention heads, embedding size 512 and intermediate size 2048, and small models composed of 4 layers, 8 attention heads, the same embedding and intermediate size of the medium ones. For SLNet we use a sequence length of 1024 tokens, thus truncating 30% of the sequences. We choose this value since it implements a reasonable trade-off between the number of truncated sequences and the computational capabilities of our hardware. The masking rate for the SLNet m_l is a value in $\{0.6, 0.8, 1.0\}$. For MapNet the sequence length is 512 tokens (99.9% of the mapping sequences are above that threshold) and the masking $m_a \in \{0.0, 0.2, 0.4\}$.

We train SLNet for 60 epochs and MapNet for 30 epochs, taking the models with the lowest validation loss. The training uses Adam optimizer with a learning rate of 10^{-5} on 8 A100 GPUs using a batch size of 16 for each device.

1) METRICS

We evaluate the performances of our model in identifying bugs by computing the Area Under the Curve (AUC). This is done by assigning to each function trace one SLNet score and one MapNet score as defined in the inference Sections IV-A2, IV-B2. On the synthetic datasets, we compute a single AUC for each class of bug; we do this by selecting a certain category (e.g., Swap Source) and ignoring all the bugged samples belonging to other categories. In this way, we compute the specific performance on a certain category of bug. On the real dataset, we use a single AUC score as bugs do not belong to a specific category.

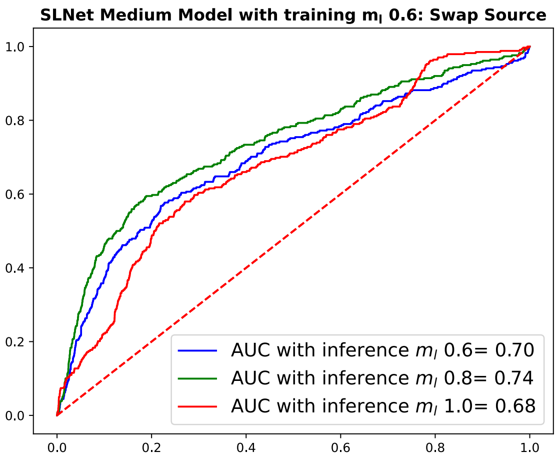

FIGURE 4. Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$.

B. RESULTS ON THE SYNTHETIC DATASETS

In this Section, we will show the results we obtained with both SLNet and MapNet on the synthetic datasets.

1) SLNET

Figure 4 shows the results of SLNet on the Swap Source bug category, which is the only bug category recognized by SLNet. In this case, the SLNet model reaches an AUC of 0.74 when using the medium model with $m_l = 0.6$ during training and $m_l = 0.8$ during inference. This is the model with the best performance; all the small models, as well as the medium models with training $m_l \in \{0.8, 1.0\}$, have worse or comparable performances.

The ROC curve for the Remove Step bug category of SLNet is in Figure 5. The performance of the network is similar to a random classifier. For the Swap Assembly bug category, the ROC is in Figure 6; also in this case the network behaves as a random classifier. We believe that SLNet provides random performance on these bugs since the removal of a step rarely impacts the source line trace; the swap of assembly instructions, that leaves intact the source line information, is invisible at the source level.

2) MAPNET

The results for MapNet are reported in Figures 7, 8, and 9. MapNet exhibits the best performance when trained with medium model, $m_a = 0.2$, and evaluated with $m_a = 0.0$. The Swap Source bug category (Figure 7) is the one with the highest results (AUC 0.89); this is expected as this category of bugs is analogous to the defects inserted in the training task. MapNet is able to discover also the

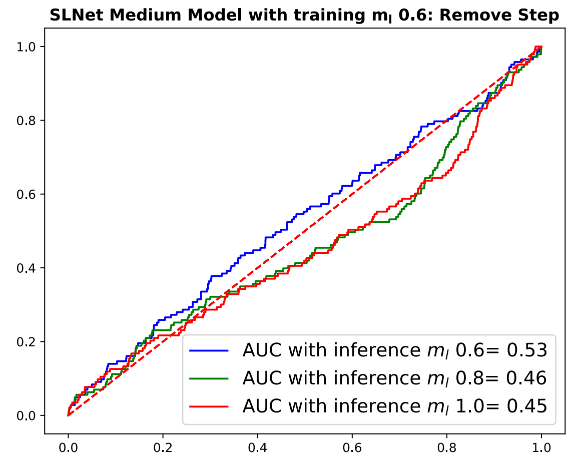

FIGURE 5. **Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$: Remove Step bug category.**

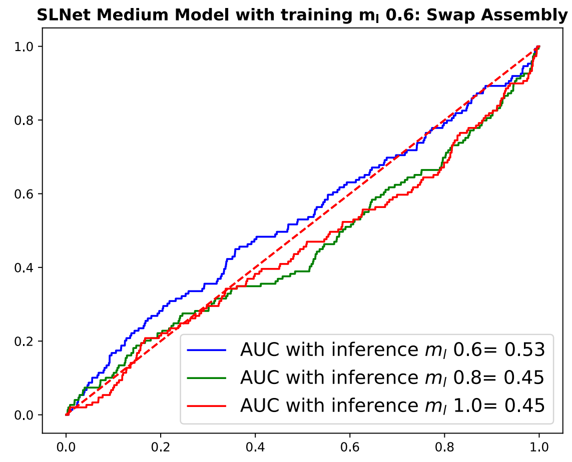

FIGURE 6. **Results on the synthetic dataset of the SLNet trained with $m_l = 0.6$: Swap Assembly bug category.**

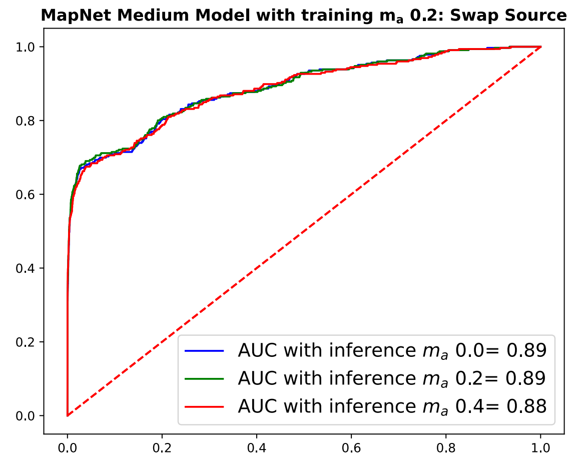

FIGURE 7. **Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Swap source bug category.**

Swap Assembly pattern (Figure 9), where a single assembly instruction is misplaced, with acceptable performance (AUC 0.75). Therefore, we expect that it will be able to identify real bugs where the sequence of assembly instructions mapped by

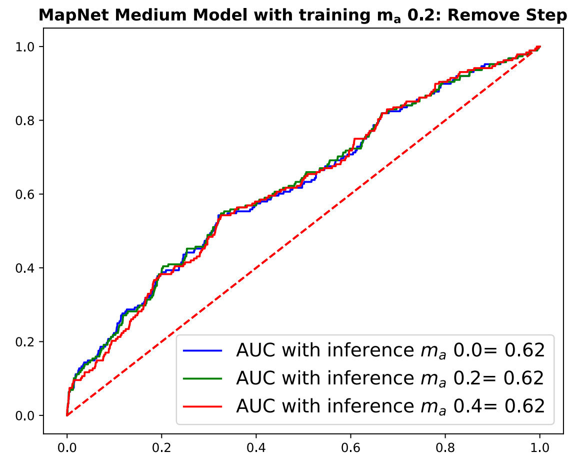

FIGURE 8. **Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Remove Step bug category.**

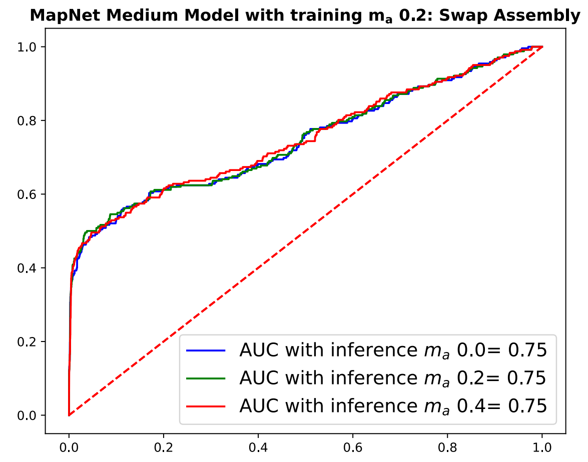

FIGURE 9. **Results on the synthetic dataset of the MapNet trained with $m_a = 0.2$: Swap assembly bug category.**

the debug info is partially correct (we will show that this is confirmed in the following sections). The worst performance is shown in the Remove Step category (Figure 8) with an AUC of 0.62. This is expected as it is far easier to recognize a completely out-of-context assembly instruction, than a missing assembly instruction. MapNet reaches a higher AUC than SLNet on the Swap Source, however, it does so by using a completely different mechanism looking at the mismatch between assembly and source line.

C. RESULTS ON THE REAL BUGS DATASETS

Given the limited size of the real dataset and the randomness of the masking procedure, we decided to increase the robustness of the results by running the inference procedure multiple times and by reporting the mean AUC value. In order to compute the results on the real dataset, we used the best models according to the synthetic dataset, which are the medium models with training $m_l = 0.6$ and $m_a = 0.2$ for the SLNet and MapNet respectively.

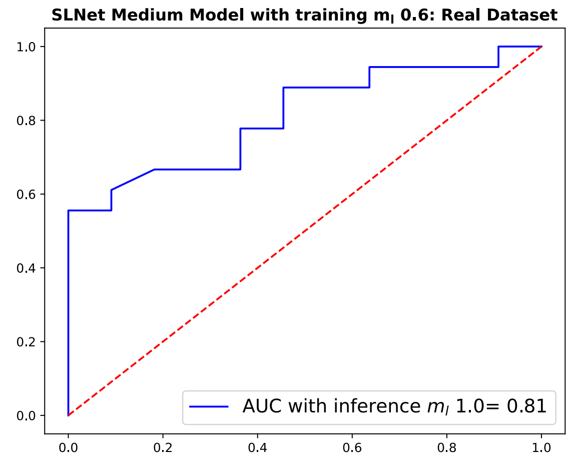

FIGURE 10. Results on real dataset of the SLNet trained with $m_l = 0.6$.

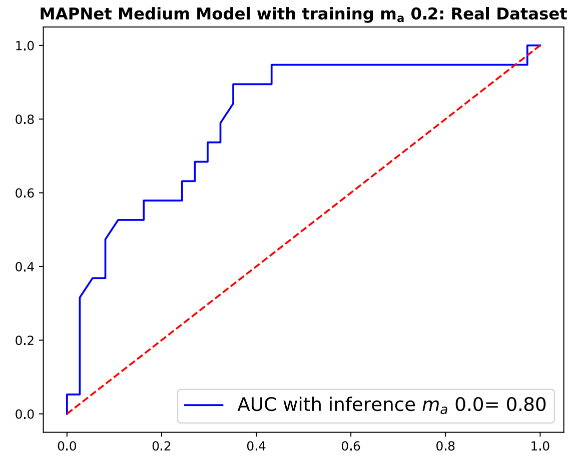

FIGURE 11. Results on real dataset of the MapNet trained with $m_a = 0.2$.

1) SLNET

During inference, we used $m_l \in \{0.6, 0.8, 1.0\}$ and, differently from the synthetic case, we obtained the best performances when $m_l = 1.0$, as shown in Figure 10. This is probably due to longer lines and higher variability for csmith source code that makes more difficult for the network to predict them without some suggestions (provided by a certain percentage of non-masked tokens). On the contrary, the real dataset contains small test cases with small lines having a lower degree of randomness, thus the network does not need suggestions to predict them. On the real dataset, SLNet reaches an AUC of 0.81.

2) MAPNET

We used $m_a \in \{0.0, 0.2, 0.4\}$ and we obtained the best performances when $m_a = 0.0$, as in the synthetic dataset case (Figure 11), the AUC reached is 0.8. This confirms that MapNet is able to identify real bugs.

D. THRESHOLD ANALYSIS

In Section VI-B, VI-C, we evaluated the performances of our networks by using the AUC metric. The aim of this Paragraph is to show the value of metrics at fixed thresholds.

TABLE 1. Precision and recall of SLNet and MapNet with fixed thresholds. The values of m_l and m_a are used during inference.

	SLNet $m_l = 0.8$	MapNet $m_a = 0.0$	
Threshold	3.3	0.5	
Bug Category	Swap Source	Swap Source	Swap Assembly
Precision	0.20	0.71	0.55
Recall	0.46	0.57	0.38
F1	0.28	0.64	0.45

TABLE 2. Precision and recall of random SLNet and MapNet with fixed thresholds.

	Random classifier for source lines	Random classifier for mapping assembly-line	
Bug Category	Swap Source	Swap Source	Swap Assembly
Precision	0.053 ± 0.003	0.028 ± 0.002	0.024 ± 0.002
Recall	0.50 ± 0.03	0.50 ± 0.027	0.50 ± 0.031
F1	0.096 ± 0.005	0.062 ± 0.003	0.047 ± 0.003

In particular, we compute the precision, recall, and F1 of the most prominent categories of bugs by using our best models, as defined in Section VI-C. We selected the thresholds to maximize the precision metrics: we want to minimize the number of false-positive bugs triggered and the waste of human time analyzing them. The chosen thresholds are 3.3 and 0.5 for SLNet and MapNet respectively. Results (see Table 1) confirm that MapNet can identify bugs with acceptable performances. In fact, MapNet reaches a precision of 0.71 and a recall of 0.57 with the Swap Source bug category, while SLNet reaches a precision of 0.20 and a recall of 0.46 on the same bug category. To have a reference, we also compute these metrics for a random classifier that uniformly flags a trace as bugged or not (see Table 2). Since these datasets are highly imbalanced, in some cases the recall is higher than the one obtained by our networks; on the contrary, precisions are very low compared to the ones shown in Table 1, thus confirming the effectiveness of our system.

E. MAPNET AND SLNET CORRELATION

We employ the data described in Section VII-A, to analyze the possibility of linear correlation between the highest losses obtained by SLNet and the ones of MapNet using Spearman's ρ [41], the results show that there is none to very-weak negative correlation (spearman -0.04 with a p-value < 0.01). This is confirmed by the Kendall Tau coefficient [42] (value of -0.02 with a p-value < 0.01). Our interpretation is that the two networks are able to identify different kinds of bugs, and this is why we keep them apart in the proposed framework.

VII. FINDING NOVEL BUGS: NEURO-DEBUG^2

We integrated our best models, SLNet medium with training $m_l = 0.6$ and MapNet medium with training $m_a = 0.2$, in the Debug^2 framework [7].

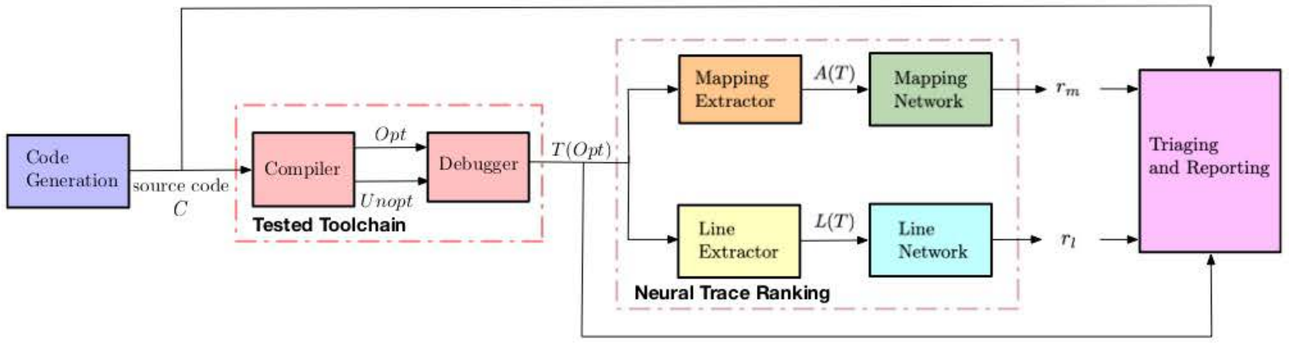

FIGURE 12. Neuro-Debug^2: System overview.

The modified framework is depicted in Figure 12 and contains the following modules:

- **Program Generation:** generates the programs that will fuzz the toolchain under test. This module uses csmith as generator. The output is a test program c.
- **Tested Toolchain:** this module contains the toolchain F to be tested. It takes as input a test program c and generates an optimized debug trace using F. Practically speaking, the input program is compiled with optimization level $-Og$ and fed into the debugger to get a set of function debug traces.
- **Neural Trace Scoring:** this module analyses each function trace T_f using the SLNet and MapNet. It outputs a tuple $< T_f, r_l, r_m >$ where r_l and r_m are the scores given by SLNet and MapNet. The Mapping Extractor and Line Extractor transform the execution trace in a sequence of source lines $L(T_f)$ and a set of mapping pairs $A(T_f)$, each object is fed into the respective network.
- **Triaging and Reporting:** this module collects the traces and orders them by their rank. The likely bugs are manually investigated; if confirmed as bugs, they are reduced to smaller examples (this is done automatically using creduce [43]). After the reduction, the confirmed likely bugs are reported to the toolchain developers.

A. TESTS AND NOVEL BUGS

We tested the LLVM version 13.0.0 of 6 April 2021. We used Neuro-Debug^2 to generate 3,983 programs, which, after preprocessing, resulted in 11,431 function traces.

1) ANALYSIS OF THE TOP SCORES

We used the data described above to compute a score for each function by using both the SLNet and the MapNet, as described in the corresponding inference Sections IV-A2, IV-B2. Our research hypothesis is that a high score for functions correlates with the presence of bugs in debug information. To validate this hypothesis we ordered all the functions by their scores (both for MapNet and SLNet) and we selected the top-scored functions.

Specifically, we analyzed the 40 top-scored functions for SLNet with inference $m_l \in \{0.6, 0.8, 1.0\}$ and MapNet with inference $m_a \in \{0.0, 0.2, 0.4\}$.[4] This leads to a total of 240 traces, 120 for each model, and 40 for each fixed combination of model/parameter. For each batch of 40 traces, we performed a manual analysis to determine how many of these traces are actually bugged or not: if a previously unknown bug was encountered during this analysis we reported it to compiler' developers.

The outcome is reported in Table 3. For SLNet, the bug percentage reaches its maximum 50% with $m_l = 1.0$ and it decreases monotonically with the masking. This means that half of the 40 top-scored functions indeed contain a bug. To rule out that this prevalence of bugs was due to chances, we took 40 random functions from our dataset and analyzed it manually noting the number of bugs encountered. From our analysis only 7% of the functions in the randomly sampled set contain a bug. This means that the prevalence of bugs in the 40 top-scored functions for SLNet is 8 times more than random.

When considering MapNet we have that 77% of the traces contain one or more bugs when $m_a = 0.4$; different masking levels give a slight decrease in performance. As for SLNet, to rule out that this effect was due to chance, we took 40 random samples from the MapNet data and analyzed them for the presence of bugs. We found that 27% of the samples contained a bug in the mapping. Note that this number is different from the 7%, reported for the analogous case in SLNet, because in this case we manually verify if each mapping is correct, while in the previous scenario we only verify that the sequence of shown source lines is correct. We highlight that also for MapNet we found almost 3 times more bugs than the ones present in a random sample. This means that highly-scored samples are likely to be bugs, thus human experts could save time in analyzing debug information correctness by using suggestions provided by our system.

2) REPORTED BUGS

From the bugs found in the previous Section, we sampled a subset that we reported to LLVM developers. Specifically,

[4]We discarded the samples that had a length above the thresholds used for truncation.

TABLE 3. Analysis of the top 40 scores for SLNet, MapNet and randomly sampled functions. The values of m_l and m_a are used during inference.

Network	Bug Percentage
SLNet $m_l = 1.0$	50%
SLNet $m_l = 0.8$	37.5%
SLNet $m_l = 0.6$	27.5%
Random traces from SLNet samples	7%
MapNet $m_a = 0.4$	77%
MapNet $m_a = 0.2$	75%
MapNet $m_a = 0.0$	72.5%
Random traces from MapNet samples	27%

we took 6 bugs found by SLNet and 6 by MapNet and we verified that they were still present in the last LLVM version 14.0.0 as of August 15th, 2021. Out of the 12 reported bugs, 2 have been confirmed by the developers; the remaining 10 bug reports are pending analysis. In this Section, we discuss one bug found by MapNet and one by SLNet.

A bug found by SLNet is in Snippet 3. In this case, lldb shows that the execution steps on line 6 (`int i, j;`); however, this is a variable declaration that should not be shown during the execution. This bug is likely due to an error in the line table (the structure that maps assembly instructions to source lines) produced by clang. We speculate that SLNet flags the stepping on a declaration as unlikely since it has not been observed as a normal behavior during training.

```
1  int a, c, e, f;
2  static int *b = &a;
3  short d = 6;
4  void func_15() {
5    for (; c >= 0; c--) {
6      int i, j;
7      *b = f;
8    }
9  }
10 int main() {
11   func_15();
12   for (; d <= 0;) {
13     int g[4];
14     g[e] = &b;
15   }
16 }
```

Snippet 3. Clang bug 51512, wrong assembly mapped to line 6.

MapNet discovered the bug in Snippet 4; in this case the call of `func_1` at line 6 is wrongly associated to an assembly instruction that set variable a to 0 (`movw $0 × 0, 0 × 200b89(rip)`); this assembly instruction should instead be,mapped to the body of `func_1`. As in the case of Snippet 1, this bug is probably the result of the inlining optimization.

VIII. COMPARISON WITH DEBUG^2 AND LIMITATIONS

In this Section, we compare our approach with Debug^2, and then we discuss the limitation of our approach.

```
1 short a;
2 int b;
3 void func_1() { a = 0; }
4 int main() {
5   b = 0;
6   func_1();
7 }
```

Snippet 4. Clang bug 51751, wrong assembly mapped to line 6.

TABLE 4. Results of the comparison between Neuro-Debug^2 and Debug^2. The values of m_l and m_a are used during inference.

Network	Neuro-Debug^2 Bugs	Debug^2 Bugs
SLNet $m_l = 1.0$	20	11
MapNet $m_a = 0.4$	31	2

A. COMPARISON WITH DEBUG^2

We evaluate the effectiveness of our proposed solution with the respect to the invariants-based approach of Debug^2 [7].

In particular, we take the 31 programs containing bugs detected by the MapNet with training $m_a = 0.2$ and inference $m_a = 0.4$ and the 20 programs containing bugs detected by the SLNet with training $m_l = 0.6$ and inference $m_l = 0.1$ (see Table 3). We measure how many times these bugged functions are identified by an invariant violation of Debug^2. We find out that, among the 31 bugs detected by MapNet, only 2 are discovered by Debug^2, while 11 out of 20 bugs discovered by SLNet were identified by Debug^2 as well (see Table 4). In our test, Debug^2 is more capable of finding bugs identified by the SLNet rather than MapNet ones. This is expected, SLNet has been designed to detect source lines that are out-of-context, some of these cases are covered by the LineInvariant of Debug^2.

We want to stress that our system is not alternative to an invariant-based approach; a user may analyze bugs found with our solution and identify a general pernicious behavior that could lead to the definition of a new invariant.

B. LIMITATIONS

1) MANUAL ANALYSIS

Confirming the presence of a bug in an anomalous trace must be done by a human expert. Table 3 quantifies how much effective the job of the expert is when working on anomalous traces identified by our system vs. random traces. The manual effort to decide if an anomalous trace is a bug or not requires around 10-20 minutes of time by a human expert. We remark that this manual analysis step is also needed by the other works that find bugs in debug information [7].

2) VARIABLE VALUES

In addition to the manual analysis requirement, another limitation of our system is represented by the absence of an explicit analysis of variable values. [7] uses two invariants which are based on variables: Scope Invariant and Parameters Invariant. The former checks whether there exists a step where a variable is visible only in the optimized trace, while the latter checks whether the optimized trace contains function parameters values that are not present in the optimized one. Currently, our system is not able to identify mistakes in the values of variables. As future work, we plan to extend our system to directly integrate them.

3) CORRECTNESS OF TRAINING DATA

As in many previous works on neural bug finding [8], [29], we assume that our training data is correct. However, we have no guarantees on such correctness. We argue that this is not a problem. Even if some bugs are so frequent in training that they are not detected as anomalous anymore, this does not jeopardize our approach. As long as there are rare bugs (that could appear in the training data but sparingly) they will be detected as anomalous. The fact that some bugs are rare is reasonable; the contrary would imply that debug information is almost meaningless. Moreover, frequent bugs are likely to be found by humans.

IX. CONCLUSION

In this paper, we introduced two DNN-based architectures trained for the detection of bugs in debug information attached to optimized binary code. Our results show that the proposed models, namely SLNet and MapNet, are capable of discovering bugs both in synthetic and real datasets. As a result of this study, 12 new bugs in the LLVM toolchain were discovered.

REFERENCES

[1] C. Jia and W. K. Chan, "Which compiler optimization options should I use for detecting data races in multithreaded programs?" in *Proc. 8th Int. Workshop Autom. Softw. Test (AST)*, May 2013, pp. 53–56.

[2] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *Proc. IEEE Secur. Privacy Workshops*, May 2015, pp. 73–87.

[3] J. Yin, G. Tan, H. Li, X. Bai, Y.-P. Wang, and S.-M. Hu, "Debugopt: Debugging fully optimized natively compiled programs using multistage instrumentation," *Sci. Comput. Program.*, vol. 169, pp. 18–32, Jan. 2019.

[4] J. Hennessy, "Symbolic debugging of optimized code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 323–344, Jul. 1982.

[5] DWARF Standards Committee. (2020). *The Dwarf Debugging Standard*. Accessed: Aug. 2021. [Online]. Available: http://dwarfstd.org/

[6] Y. Li, S. Ding, Q. Zhang, and D. Italiano, "Debug information validation for optimized code," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2020, pp. 1052–1065.

[7] G. A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni, "Who's debugging the debuggers? Exposing debug information bugs in optimized binaries," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 1034–1045.

[8] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–17.

[9] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2613–2630.

[10] J. Xu, K. Lu, and B. Mao, "Cross-architecture testing for compiler-introduced security bugs," in *Proc. 5th Workshop Princ. Secure Compilation (PriSC)*, 2021.

[11] H. Lim and S. Debray, "Automated bug localization in JIT compilers," in *Proc. 17th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Apr. 2021, pp. 153–164.

[12] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 335–346.

[13] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 197–208.

[14] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 968–980.

[15] J. Chen, "Learning to accelerate compiler testing," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 472–475.

[16] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, Jan. 2021.

[17] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2011, pp. 283–294.

[18] D. Babokin, J. Regehr, and V. Livinskiy. (2020). *YARPGen: Yet Another Random Program Generator*. Accessed: Jul. 27, 2020. [Online]. Available: https://github.com/intel/yarpgen

[19] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, Oct. 2015, pp. 386–399.

[20] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2014, pp. 216–226.

[21] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proc. 27th Int. Conf. Compiler Construct.*, Feb. 2018, pp. 82–92.

[22] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," 2020, *arXiv:2002.12543*.

[23] M. Allamanis, T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 81, pp. 1–37, 2018.

[24] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2018, pp. 95–105.

[25] X. Liu, X. Li, R. Prajapati, and D. Wu, "DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing," in *Proc. 33th AAAI Conf. Artif. Intell.*, 2019, pp. 1–8.

[26] D. Lehmann and M. Pradel, "Feedback-directed differential testing of interactive debuggers," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 610–620.

[27] S. Tolksdorf, D. Lehmann, and M. Pradel, "Interactive metamorphic testing of debuggers," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 273–283.

[28] R. Gupta, S. Pal, A. Kanad, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning," in *Proc. 31th AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351.

[29] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," 2019, *arXiv:1904.01720*.

[30] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proc. 5th ACM SIGPLAN Int. Symp. Mach. Program.*, Jun. 2021, pp. 1–8.

[31] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proc. 38th Int. Conf. Mach. Learn.*, vol. 139, 2021, pp. 780–791.

[32] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, Oct. 2016, pp. 849–863.

[33] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2012, pp. 78–88.

[34] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 2016, pp. 70–81.

[35] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2015, pp. 65–76.

[36] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020, *arXiv:2004.05150*.
[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[38] G. Lample and A. Conneau, "Cross-lingual language model pretraining," in *Proc. 33th Conf. Neural Inf. Process. Syst.*, 2019, pp. 1–10.
[39] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng. (ICSE)*, Jun. 2020, pp. 1073–1085.
[40] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: On the importance of pre-training compact models," 2019, *arXiv:1908.08962*.
[41] G. U. Yule, *An Introduction to the Theory of Statistics*. London, U.K.: Charles Griffin and Company, 1911.
[42] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1, pp. 81–93, Jun. 1938.
[43] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*. New York, NY, USA: ACM, Jun. 2012, pp. 335–346.

FIORELLA ARTUSO received the master's degree in engineering in computer science, in 2019. She is currently pursuing the Ph.D. degree in engineering in computer science with the Sapienza University of Rome. During 2020, she worked as a Researcher with the Consorzio Interuniversitario Nazionale per l'Informatica (CINI). Her research interests include the application of deep learning and natural language processing techniques to source or binary code to solve problems in the software engineering and cyber security fields.

GIUSEPPE ANTONIO DI LUNA received the Ph.D. degree from the Sapienza University of Rome, in 2015, with a thesis on counting in anonymous dynamic networks. After his Ph.D. degree, he did a postdoctoral research at the University of Ottawa, working on fault-tolerant distributed algorithms, distributed robotics, and algorithm design for programmable particles. In 2018, he started a postdoctoral research at Aix-Marseille University, where he worked on dynamic graphs. Currently, he is working with the Sapienza University of Rome, performing research on applying natural language processing techniques to the binary analysis domain.

LEONARDO QUERZONI received the Ph.D. degree, in 2007, with a thesis on efficient data routing algorithms for publish/subscribe middleware systems. He is an Associate Professor with the Sapienza University of Rome. He has authored more than 80 papers published in international scientific journals and conferences. In 2016, he has coauthored the *Italian National Framework for Cyber Security* as a member of the Cyber Intelligence and Information Security Research Center, Sapienza University of Rome. His research interests include a range from cyber security to distributed systems and focus, in particular, on topics that include binary analysis, distributed stream processing, dependability, and security in distributed systems. In 2017, he received the Test of Time Award from the ACM International Conference on Distributed Event-Based Systems for the paper *TERA: Topic-Based Event Routing for Peer-to-Peer Architectures* (2007). In 2014, he was the General Chair of the International Conference on Principles of Distributed Systems and was the Program Co-Chair of the ACM International Conference on Distributed Event-Based Systems, in 2019.