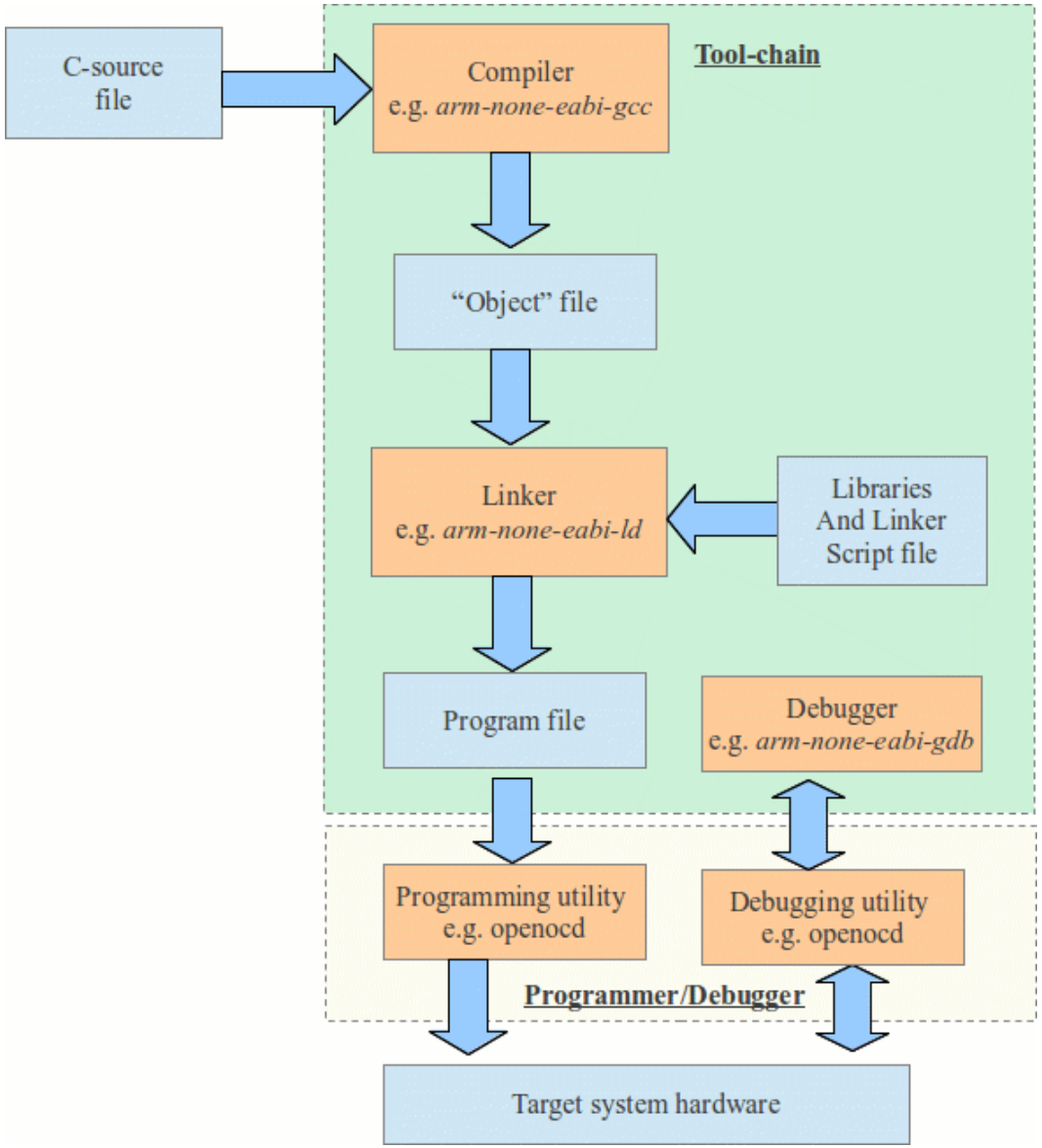
Debugging Debug Information with Neural Networks

전자전기컴퓨터공학부
2022440119 전형준

Index



Compiler Toolchain

소스 코드를 실행 가능한 프로그램으로 변환하고
디버깅을 가능하도록 하는 개발 도구 집합

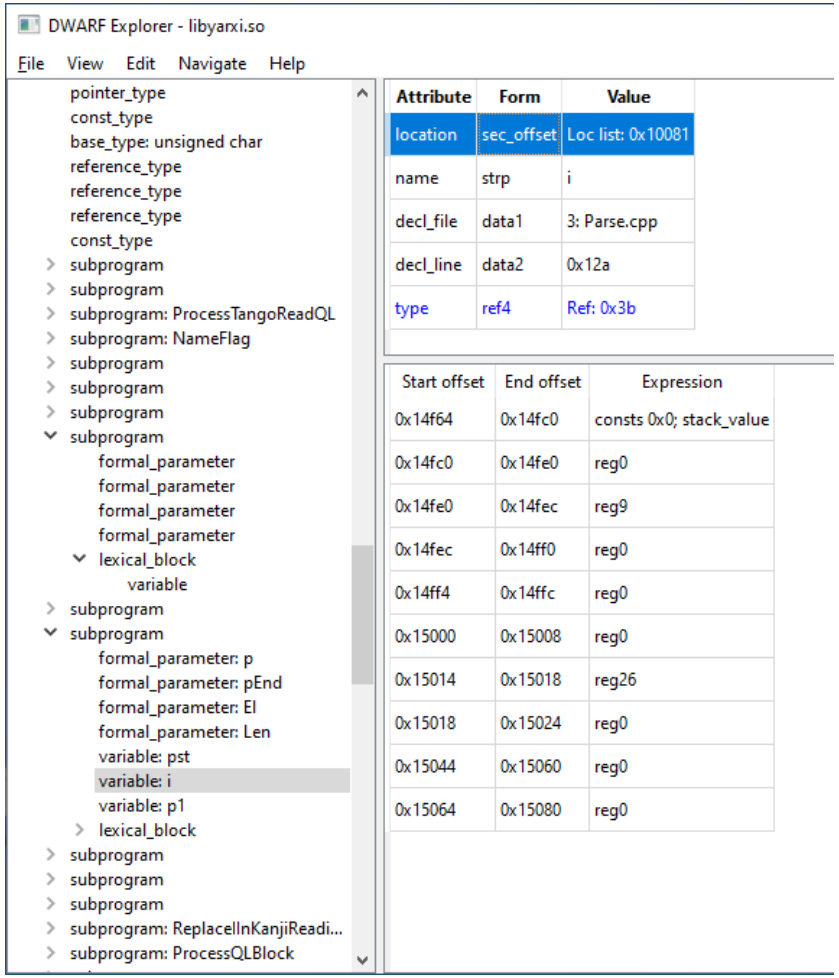
Compiler, Linker, Debugger, Profiler 등등이 사용됨.

https://ioprog.com/arm-cortex/the-toolchain/

Debug Information

Debug information은 프로그램이 컴파일 된 이후에
디버깅에 사용되는 정보이다.

변수 정보: 변수의 이름, 타입, 저장 위치
함수 정보: 함수의 이름, 시작 및 종료 주소, 매개변수 등의 정보
소스 코드 매핑: object code와 source 코드 사이의 mapping 정보를 저장



Compiler Optimization

컴파일러를 사용하여 목적 코드(이진 코드)를
생성하는 과정에서 효율성을 위해 최적화를 진행함.
사용자의 목적에 따라서 최적화 정도를 조절할 수 있음.

논문에서 예시로 들고 있는 GCC / Clang의 경우
-O0 (최적화 없음), -O1, -O2, -O3 (가장 공격적인 최적화),
-Os (크기 최적화), -Og (디버깅을 위한 최적화) 등이 존재

```c
int a = 1;
int b = 2;
int c = 3;
int result = a + b + c;
```

최적화 없이 컴파일 된 경우

```assembly
mov eax, 1        : a = 1
mov ebx, 2        : b = 2
mov ecx, 3        : c = 3
add eax, ebx      : a = a + b
add eax, ecx      : a = a + c
mov result, eax   : result = a
```

가장 높은 수준의 최적화를 적용한 경우

```assembly
mov eax, 6        ; x = 1 + 2 + 3
mov result, eax   ; result = x
```

Compiler Optimization

높은 수준의 최적화를 진행하는 경우, 기존의 source 코드와
생성되는 object 코드 간의 차이가 심해지기에, debug information의
정확성을 유지하는 것은 굉장히 어렵다.

Debug information 정보의 정확성을 최대한 유지하기 위해서
-Og 옵션을(디버깅을 위한 옵션) 사용한다.

Debugging
Debug Information

컴파일 과정에서 디버깅을 위한 옵션 –Og를 사용해도,
Debug information의 정확도가 낮은 문제가 발생함.

이는 Toolchain 자체의 문제이므로, 어떤 문제가 발생하는지 확인하기 위해서는 Debug Trace 정보가 필요하다.

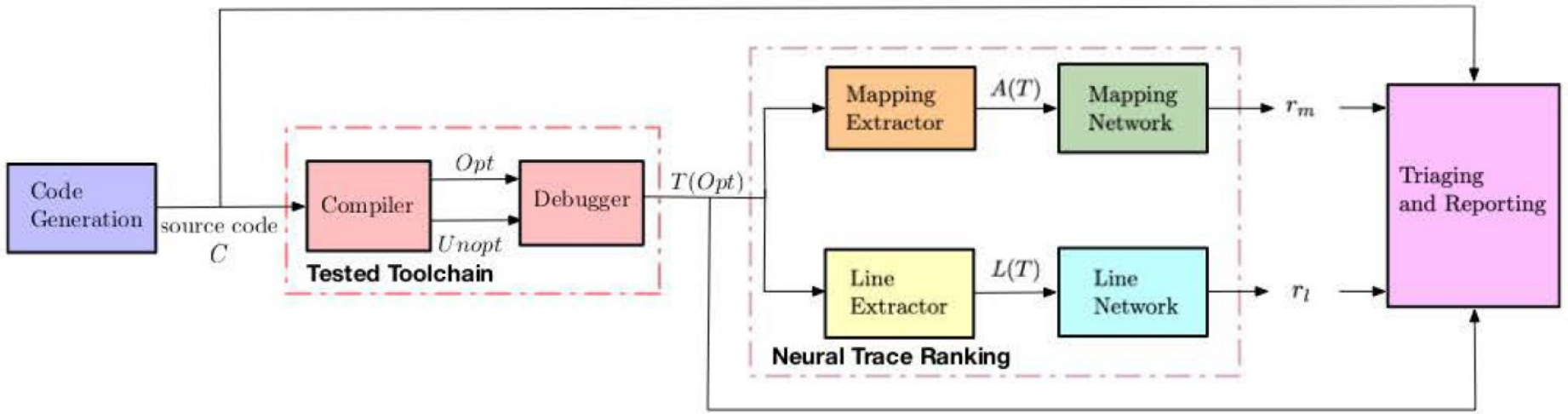
```
1  volatile int a,  g_5108; int b;
2  short c(){
3   return g_5108;
4  }
5  int main () {
6   c();
7   b = 0;
8   for (;
9   b < 4;
10  b++) a ;
11 }
```

Debugging
Debug Information

기존 Toolchain의 검증 과정은 optimization을 진행하지 않은 코드와 진행한 코드의 debug trace를 비교하여 불변식이 일치하지 않은 경우를 식별하는 방식으로 진행되었기에, 개발자가 직접 정의해야 했다.

불변식(검사 요소)을 개발자가 직접 정의해서 잘못된 debug trace를 찾아내는 것이 아니라 Unsupervised Learning을 통해서 해결하는 것이 이번 논문의 주요 목표이다.

Debugging
Debug Information



Source Code Generation

Toolchain에서 compiler나 Debugger의 오류로 인한 영향이 없다고 간주하기 위해서 Csmith로 생성된 프로그램을 사용했다.

Csmith로 생성된 프로그램은 정의되지 않은 동작이 없기에 Debug Trace가 의미 있는 정보를 제공하고 실제 소프트웨어 동작을 정확하게 반영하도록 보장한다.

Debug trace 생성

*C*를 source code로 가정하자.
Opt는 컴파일러가 c를 입력으로 받아 최적화 기법을 적용하여
생성한 기계어 code 프로그램이다.

생성된 opt 프로그램에 대해서 디버거를 통해 기계어 명령어를 단계별로 실행하며 생성되는 실행 경로 로그가 debug trace *T(opt)*에 해당한다.

T(opt) = [s0, s1, s2, … , sn] 형태로 정의할 수 있다.

Debug trace 생성

T(opt)의 s0, s1 등등의 각 step에 대해서 2가지 정보를 수집하여 활용하는데, 이는 line(s)와 asm(s)이다.

Line(s)는 현재 step에서 assembly 코드의 실행으로 동작하는 c 코드의 위치를 의미하고,
Asm(s)은 현재 step에서 실행되는 assembly 코드의 위치를 의미한다.

Debug trace 생성

만약 source code c가 여러 함수로 구성되어 있다면,
C = {f1, f2, … fm}으로 표기할 수 있고,
Trace 정보 또한, 해당 함수들 단위로 나누어 표기할 수 있다.
T(opt) = Tf1, Tf2, …, Tfm

$$\text{opt} = \text{Compile}(c)$$

$$T(\text{opt}) = \text{Debug}(\text{opt})$$

$$T_{f_j} = \text{Extract}(T(\text{opt}), f_j)$$

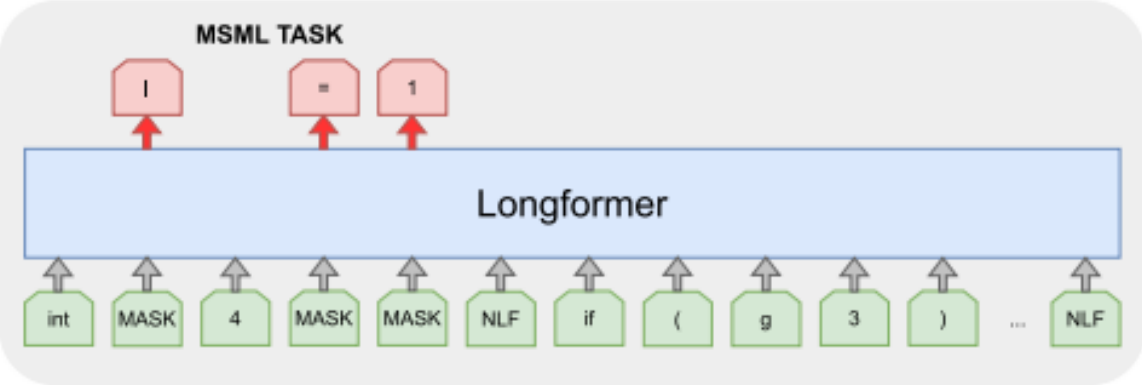
Model Input data for SLNET

L(Tf) – Sequence of source Line

L(Tf)는 특정 함수에 대한 Debug trace에 속하는 step의 소스 코드 라인을 순서대로 나열한 시퀀스에 해당한다. 즉, 특정 함수를 실행시켰을 때 실행되는 소스 코드 라인을 연속적으로 저장한 형태이다.

$$L(T_f) = [l_1, l_2, \ldots, l_k]$$

Training SLNet

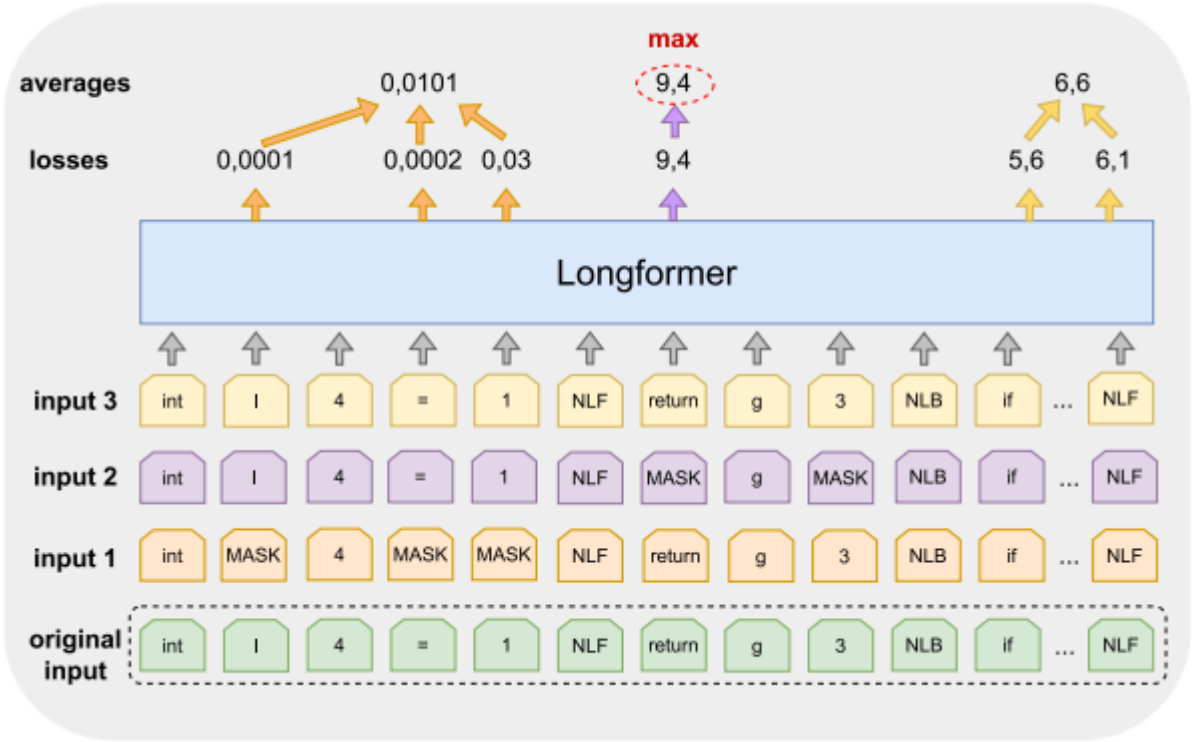


SLNet은 Masked Source Language Modeling Task를 수행하도록 학습되었다.

Input의 특정 비율의 토큰을 Masking token으로 변환한 뒤, 해당 Masking을 예측하도록 하는 방식으로 학습을 진행함.

모델의 예측값과 실제 token들의 차이가 크다면, 현재 라인의 코드가 잘못된 문맥에서 사용되고 있다고 간주한다.

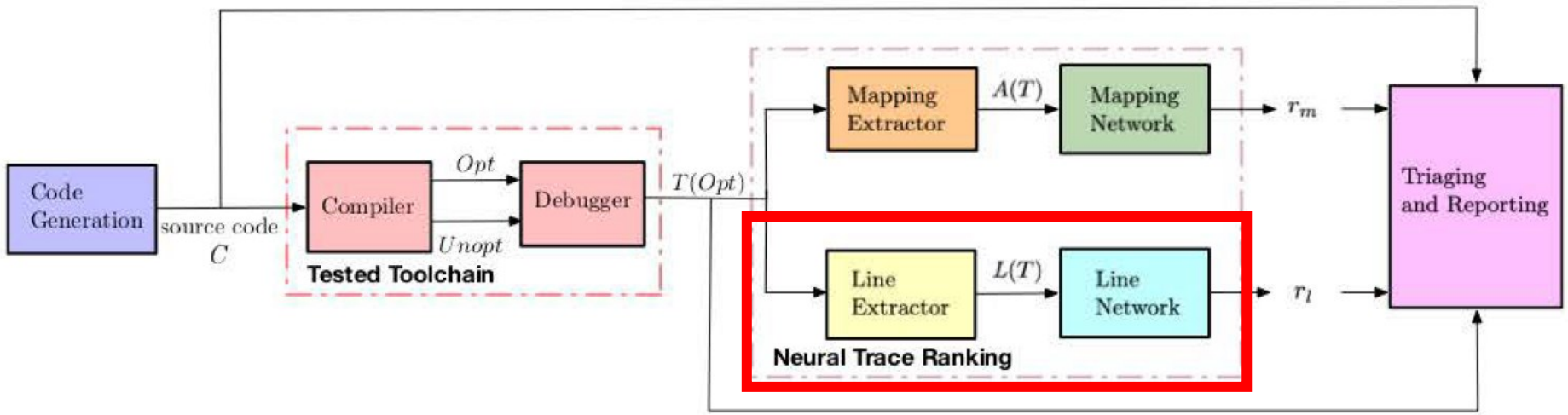
Training SLNet



특정 함수에서 실행되는 코드 라인 별로,
각 라인의 모든 masking값의 예측 token과 실제 token의 차이를 평균하여 각 라인 별 오차를 구하게 된다.

각 라인의 오차들 중에서 최댓값을 해당 함수의 점수로 갖고, 여러 함수들 중에서 높은 점수를 뽑아 debug information의 오류 의심 함수를 결정함.

방법론


Code Generation
source code
C
Compiler
Opt
Unopt
Debugger
Tested Toolchain
T(Opt)
Mapping Extractor
A(T)
Mapping Network
r_m
Line Extractor
L(T)
Line Network
r_l
Neural Trace Ranking
Triaging and Reporting

Model Input data for MapNet

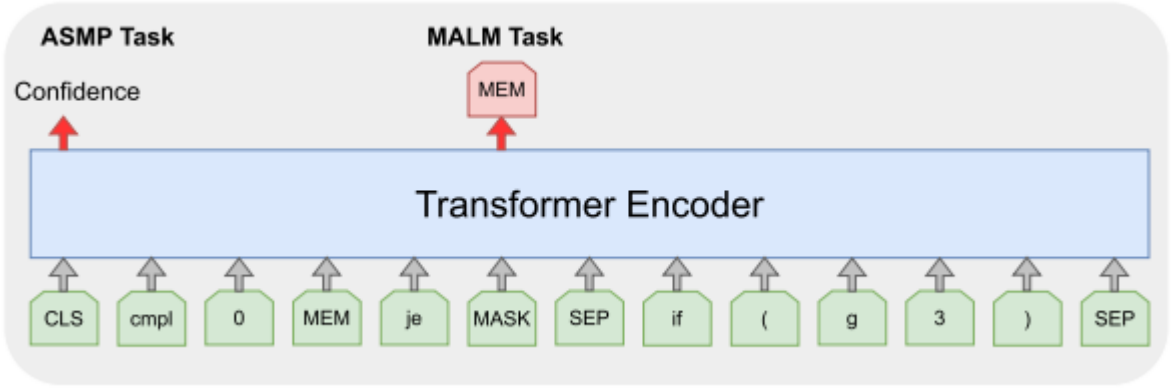
MapNet에서는 어셈블리 언어와 원본 코드의 Mapping과 어셈블리 언어의 문맥을 학습하도록 하는 것을 목적으로 함.

MapNet에서 사용되는 데이터는 각 함수별로 실행되는 source line과 매핑된 assembly code 정보들이다.

특정 함수의 debug trace Tf에 대해서 MapNet에서 사용되는 데이터는 A(Tf)로 표기하는데, 이는 아래와 같다.

$$A(T_f) = \{([a_0, \ldots, a_m], l) \mid \text{each } a_i \text{ is an assembly instruction mapped to } l \text{ in } T_f\}$$

Training MapNet

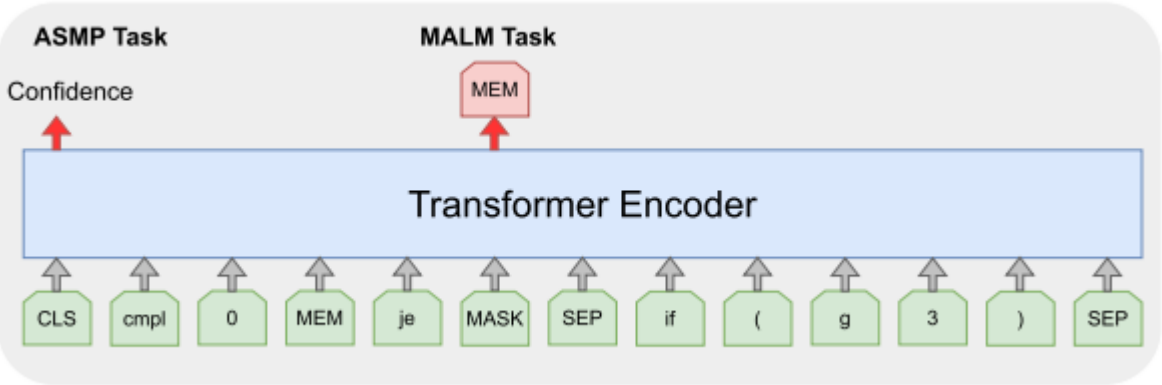


MapNet Training
1. Asm/Source Mapping Prediction
2. Masked Asm Language Modeling

1. Asm/Source Mapping Prediction
기존 Bert model의 NSP(Next Sentence Prediction)을 활용하여 Assembly code – Source code Mapping을 학습함.

정답 : A(Tf)
오답: A(Tf) + Permutation
Binary Classification 문제를 학습

Training MapNet

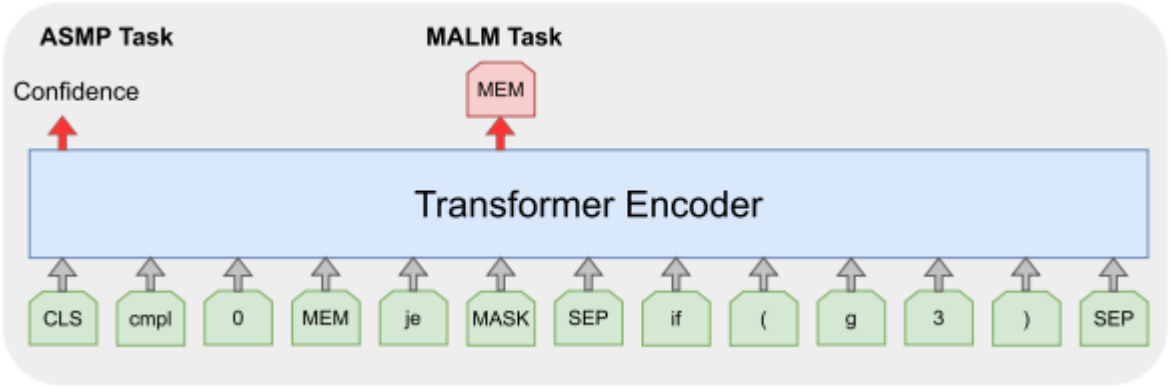


MapNet Training
1. Asm/Source Mapping Prediction
2. Masked Asm Language Modeling

2. Masked Asm Language Modeling
이전 SLNet에서 학습시킨 것과 동일하게, Assembly 언어에 대해서도 Masking을 진행하고, 해당 Masking 값을 training하는 과정으로 Assembly 작동 방식에 대해서도 학습을 진행함.

training loss는 1과 2의 sum으로 정의됨.

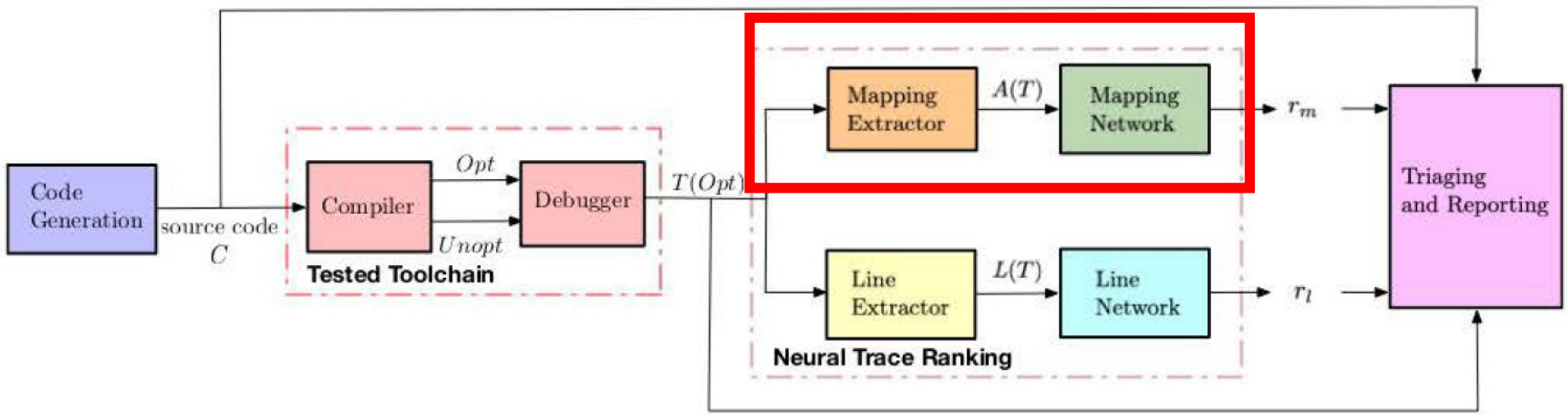
Training MapNet



MapNet Inference
ASMP에 대한 평가 지표만 사용

Bert model의 confidence score를 통해서
wrong mapping에 대한 확률을 구할 수 있고,
A(Tf)에 해당하는 모든 조합에 대해서 가장 최댓값을
각 함수의 score로 가진다.

방법론


Code Generation
source code
C
Compiler
Opt
Unopt
Debugger
Tested Toolchain
T(Opt)
Mapping Extractor
A(T)
Mapping Network
r_m
Line Extractor
L(T)
Line Network
r_l
Neural Trace Ranking
Triaging and Reporting

Dataset

SLNet, MapNet Train Dataset – Csmith Data

Detail	SLNet Dataset	MapNet Dataset
Total Programs	43,921 (common)	43,921 (common)
Debug Function Traces	200,443 (common)	200,443 (common)
Training Dataset	81,337 traces	349,153 mapping pairs
Validation Dataset	7,195 traces	25,286 mapping pairs
Dataset Split	90% training, 10% validation	90% training, 10% validation
Tokens in 70% of Samples	Less than 1,024 tokens	
Tokens in 99.9% of Samples		Less than 512 tokens

Dataset

SLNet, MapNet Evaluation Dataset – Synthesis Data

Dataset	Non-Bugged Samples	Bugged Samples	Swap Source	Swap Assembly	Remove Step
SLNet Dataset	6,009	630	338	149	143
MapNet Dataset	90,739	10,653	5,352	3,093	2,208

Swap Source: 두 스텝 사이에서 소스 라인을 임의로 교체
Swap Assembly: 두 스텝 사이에서 어셈블리 명령을 임의로 교체
Remove Step: 하나의 스텝을 임의로 제거

Dataset

SLNet, MapNet Evaluation Dataset – Real Bug data

Dataset	Total Traces/ Samples	Bugged	Bug-Free
SLNet Dataset	29 traces	18	11
MapNet Dataset	136 samples	76	60

LLVM Storage에서 보고된 실제 버그 42건에 대해서 잘못된
Mapping을 보이는 사례들을 식별해 데이터를 생성

Bugged는 Bug가 포함된 Toolchain 버전을 Bug-Free는 해당
Bug가 수정된 Toolchain 버전을 사용해 데이터를 생성

Evaluation Result

Model	Dataset Type	Evaluated Bug Type	Best AUC Score	Observations
SLNet	Synthetic	Swap Source	0.74	Best performance in identifying swap source bugs.
SLNet	Synthetic	Remove Step	Random classifier level	Poor performance on remove step.
SLNet	Synthetic	Swap Sssembly	Random classifier level	Poor performance on swap assembly.
MapNet	Synthetic	Swap Source	0.89	High accuracy in mapping bugs relevant to training scenarios.
MapNet	Synthetic	Swap Assembly	0.75	Good performance, useful for identifying real bugs.
MapNet	Synthetic	Remove Step	0.62	Lower performance, but better than random.
SLNet	Real	General	0.81	Strong performance on real bug dataset.
MapNet	Real	General	0.80	Consistently strong across real bug scenarios.

Evaluation Result

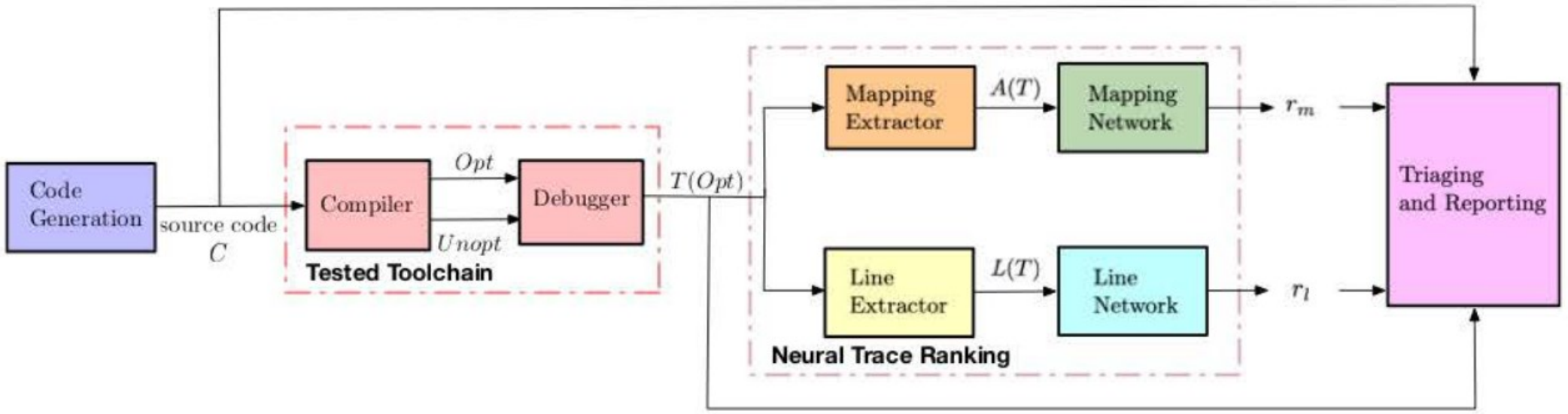


FIGURE 12. Neuro-Debug^2: System overview.

감사합니다.