Additional Technical Details for "ReGraph: Re-optimization and Lifting for Binary Function Identification Leveraging CPGs and GNNs"

Li Zhou, Marc Dacier, and Charalambos Konstantinou

King Abdullah Univerisity of Science and Technology,
`{firstname.lastname}@kaust.edu.sa`

1 Complexity in Computation Theory

The time complexity of a single self-attention layer is $O(N^2d + d^2N)$ [4], where N represents the number of tokens, and d stands for the hidden layer size, a manually set value. New models divide N tokens into m blocks. The model calculates embeddings for each code block separately, resulting in a time complexity of $O(\frac{N^2d}{m^2} + \frac{Nd^2}{m})$ for each block. Given the presence of m code blocks, the overall time complexity of the model is reduced to $O(\frac{N^2d}{m} + Nd^2)$.

In this paper, we directly employ GNNs to process CPGs. Considering the number of tokens as N, the time complexity of GNNs, such as GAT, is $O(VF'F+EF'F)$, where V represents the number of nodes in the CPG (which is smaller than N), E denotes the number of edges in the CPG (greater than V), F' denotes the length of node features (set to 150 in our case), and F means the output hidden layer size (we set it to be 128).

Taking the OpenPLC program as an example, the average token count, N for its functions, is 1733, with a maximum of 13294. After decompilation and conversion to pseudo-C code, the average function node count V for OpenPLC is 269, with a maximum of 897. The average edge count E is 599, with a maximum of 2354. NLP models incorporate multiple stacked transformers for enhanced performance [1]. For instance, the RoBERTa [2] model utilized in [3] comprises 6 transformer layers, each consisting of 8 attention layers. In contrast, our model only contains a single GAT layer. Consequently, rough theoretical calculations indicate that the computational complexity of RobertA is approximately 2542 times that of GAT. Hence, employing GNNs directly results in significantly lower time complexity and computational resource requirements than natural language models.

2 Hyper-Parameters

Please refer to Table 1

3 Applicability of the Tool Chain

While using Lifter and LLVM2CPG, we encountered some issues that required fixing. First, there are problems with function re-optimization. Due to dynamic

Table 1: The hyper-parameters chosen to train the model.

Name	Value
d_1	151
d_2	128
d_v	64
P	50
E	100
ι	0.00086
batch_size	16

linking mechanisms in the operating system, some function calls appeared as simple calls without actual addresses. When we lifted these functions, the lack of debug information caused the system to recognize these calls as regular functions. During subsequent optimization steps, the optimizer mistakenly treated these functions as meaningless jumps in data flow analysis. As a result, the relevant data flow was discarded, rendering the entire function meaningless. Code 1.1 illustrates an example where the library function `rand` is compiled as a simple call. However, because the lifter did not represent it as a library function but rather as an empty function, shown in Code 1.2, it influenced the optimizer's data flow analysis in the re-optimization phase. The optimizer considered the entire data flow to be empty, resulting in an optimized function that also became empty, as shown in Code 1.3.

```c
int test(int a1)
{
    for (int i=0; i<0x100; i++){
        x += 0x4321;
    }
    for (int i=0; i<3; i++){
        x ^= rand();
    }
    return x;
}
```

Code 1.1: The source code

```c
int32_t test(int32_t a1)
{
    int32_t result = a1 + 0x432100;
    for (int32_t i=0; i<3; i++) {
        return ^= function_114();
    }
    return result;
}
```

Code 1.2: The lifted code

```c
int32_t test(int32_t a1)
{
    int32_t result;
    return result;
}
```

Code 1.3: The re-optimized code

Furthermore, the LLVM IR obtained after using lifter also exhibited poor quality. When we passed the LLVM IR obtained from the lifter to the OPT tool for re-optimization, we sometimes encountered errors related to unordered "uselistorder," indicating incorrect function references. However, since these errors are related to debugging information, we were able to manually remove unnecessary debug information and continue using OPT for optimization.

LLVM2CPG also faced difficulties processing long functions. During our experiments, when confronted with a long binary file with more than 70 functions in our OpenPLC dataset, the entire tool would crash due to code defects in LLVM2CPG. Consequently, we had to split the source code files into smaller parts to obtain the required functions' CPG for training. However, even when dealing with shorter binaries with only 5 functions, LLVM2CPG would still crash for inexplicable reasons. Considering that this tool is no longer being actively maintained, there is little hope that someone will fix these issues. This explains why we have decided not to use it.

4 Performance Between LLVM IR and C Code

LLVM IR can be converted to CPG. Alternatively, LLVM IR can be transformed into pseudo-C code, which then can be converted to CPG. Which approach yields better performance in terms of CPG generation? To answer this question, we conducted experiments on the OpenPLC dataset, comparing the effectiveness of CPG generated from pseudo-C code and CPG derived from LLVM IR. The experimental results are presented in Table 2. It can be observed that CPG generated from C code outperforms the CPG derived from LLVM IR. We attribute this to the single static assignment properties of LLVM IR, where variables are assigned only once, and there are limited data and control flow structures. As shown in Table 2, LLVM IR code tends to generate a worse performance compared to pseudo-C code. Simple computations on a single variable in C code may be assigned to multiple different variables in the corresponding LLVM IR code, increasing the complexity and the size of the CPG. On the other hand, decompiled C code derived from LLVM IR exhibits different control structures and data structures, providing enhanced readability and better capturing of semantic information. Hence, we decided to build CPG based on further decompiled pseudo-C code instead of LLVM IR.

5 CPGs Before and After Re-Optimization

Table 2: The performance between the CPG generated from pseudo-C Code and IR.

	C	IR
Accuracy	93.53%	88.85%
Diff	0.7805	0.6461

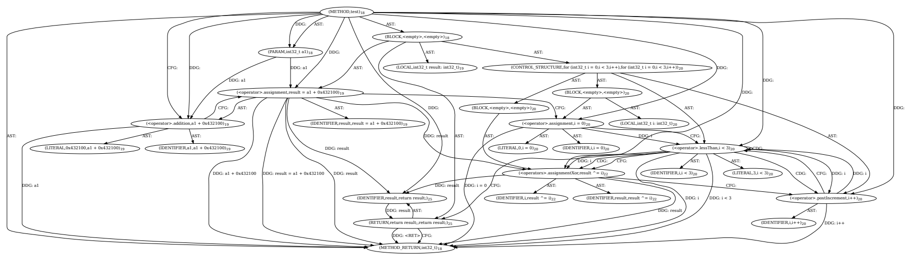

Fig. 1: The CPG before re-optimization of ARM -O0.

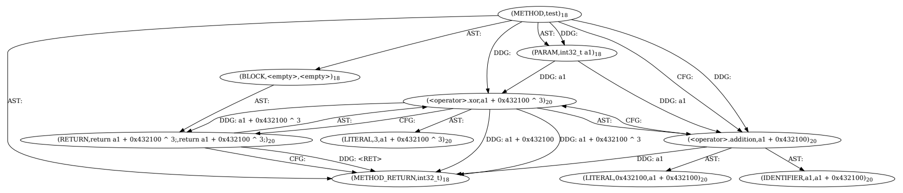
(METHOD,test)18
AST:
DDG:
(PARAM,int32_t a1)18
(BLOCK,<empty>,<empty>)18
DDG: a1
CFG:
(<operator>.xor,a1 + 0x432100 ^ 3)20
DDG: a1 + 0x432100 ^ 3
(RETURN,return a1 + 0x432100 ^ 3;,return a1 + 0x432100 ^ 3;)20
(LITERAL,3,a1 + 0x432100 ^ 3)20
DDG: a1 + 0x432100
DDG: a1 + 0x432100 ^ 3
(<operator>.addition,a1 + 0x432100)20
DDG: <RET>
(METHOD_RETURN,int32_t)18
(LITERAL,0x432100,a1 + 0x432100)20
(IDENTIFIER,a1,a1 + 0x432100)20

Fig. 2: The CPG after re-optimization of ARM -O0.

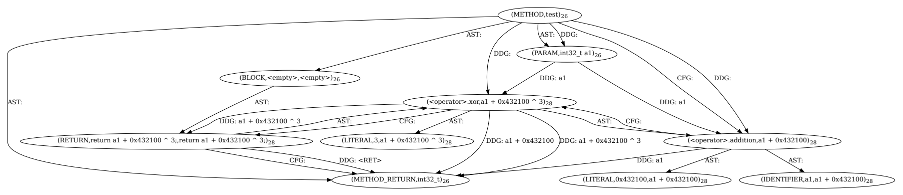
(METHOD,test)26
AST:
DDG:
(PARAM,int32_t a1)26
(BLOCK,<empty>,<empty>)26
DDG: a1
CFG:
(<operator>.xor,a1 + 0x432100 ^ 3)28
DDG: a1 + 0x432100 ^ 3
(RETURN,return a1 + 0x432100 ^ 3;,return a1 + 0x432100 ^ 3;)28
(LITERAL,3,a1 + 0x432100 ^ 3)28
DDG: a1 + 0x432100
DDG: a1 + 0x432100 ^ 3
(<operator>.addition,a1 + 0x432100)28
DDG: <RET>
(METHOD_RETURN,int32_t)26
(LITERAL,0x432100,a1 + 0x432100)28
(IDENTIFIER,a1,a1 + 0x432100)28

Fig. 3: The CPG before re-optimization of x86 -O3.

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[2] Zhuang Liu, Wayne Lin, Ya Shi, and Jun Zhao. 2021. A robustly optimized BERT pre-training approach with post-training. In *China National Conf. on Chinese Computational Linguistics*. Springer, 471–484.

[3] Zhenhao Luo et al. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS*.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).