ALFREDO: AGENTIC LLM-BASED FRAMEWORK FOR CODE DEOBFUSCATION

Ching Yuhui Natalie[1], Sophie Tung Xuan Ying[2], Siow Jing Kai[3]
[1] River Valley High School, 6 Boon Lay Avenue, Singapore 649961
[2] Nanyang Girls' High School, 2 Linden Drive, Singapore 288683
[3] DSO National Laboratories, 12 Science Park Drive, Singapore 118225

Abstract

Code obfuscation poses a significant challenge in cybersecurity, preventing reverse engineering and the effective analysis of malware. Sophisticated obfuscation tools are becoming increasingly accessible, but current deobfuscation tools remain manual and tedious to use. The few automatic deobfuscators available are often limited to one transformation, whereas in real-life contexts, multiple transformations are used. Given their versatility and efficiency, recent studies have concluded that Large Language Models (LLMs) are a promising approach to deobfuscation tasks. Novel agentic approaches can further enhance their potential, by helping LLMs leverage compilation tools such as Ghidra analysis and control decision-making.

In this work, we propose ALFREDO, an Agentic LLM-Based Framework for Code Deobfuscation. We implement a multiclass classifier and deobfuscation through LLMs, using an iterative process and tooling with agentic AI. Experiments on two datasets of Tigress-obfuscated C programs prove this framework an accurate and promising method for code deobfuscation, achieving a success rate of 77.9%. We demonstrate the versatility of LLMs in code deobfuscation, able to handle four different transformations without being constrained to just one like traditional deobfuscators.

With a framework designed for agentic systems, we lay the groundwork for equipping LLMs with more sophisticated tools, improving the effectiveness of LLM-based deobfuscation.

1 INTRODUCTION

1.1 Background

Malware employs a variety of obfuscation techniques to evade identification or analysis, making deobfuscation a significant challenge in cybersecurity. Deobfuscation refers to the process of removing obfuscation – a way of purposefully concealing or distorting parts of code to make the program difficult to detect, tamper with, or reverse engineer. It is important to note however, that obfuscation is essential in safeguarding important software against attackers in the same way that malicious actors use obfuscation to obscure malicious code and make it difficult to understand. In this aspect, obfuscation is a double edged sword, and deobfuscation tools, in the wrong hands, can potentially be used for malicious purposes.

Deobfuscation can be done either manually through the use of tools such as Miasm [16] and Ghidra [17], or computationally, such as through semantic analysis and machine learning. Manual deobfuscation is tedious and requires a lot of manual work - analysts have to perform a

multitude of static and dynamic analysis techniques. On the other hand, machine learning has been on the rise due to its ability to process vast amounts of data and replicate or even surpass human results in a shorter time. Moreover, due to its versatility, it can be applied to a vast number of practical situations. In particular, machine learning techniques for deobfuscation have gained popularity due to their potential in automating the tedious process and their ability to be trained toward a specific deobfuscation task or procedure.

1.2 Motivations of research

Large Language Models (LLMs) have gained immense popularity in recent years thanks to their versatility and potential for automation. Given LLMs' remarkable ability to generate, interpret and summarise code, they have shown promise for software engineering tasks [18], including malware analysis and deobfuscation tasks. LLMs can analyse large amounts of data from cybersecurity news articles, blogs, forums, and research papers to identify various obfuscation patterns and understand how to deobfuscate them. In their work, Pataskis et al. assess LLMs in malicious code deobfuscation from real-world malware campaigns [1]. They found that while traditional LLMs are not yet robust enough to fully replace traditional deobfuscators, they can efficiently complement them whenever they fail, and have the potential to be used independently in the future.

Traditional LLMs have faced criticism for their tendency to generate unreliable and inaccurate information, making them unpreferred for more technical tasks [2]. However, the emerging use of agentic LLM systems have shown promising countermeasures by allowing LLMs to exercise control over decision-making and leverage tools such as decompilers, as well as dissect and organise tasks. We hypothesise that with the help of agentic frameworks, LLMs can demonstrate potential in tackling code deobfuscation problems and pose a viable alternative to methodical deobfuscators.

In short, we contribute:

- ALFREDO, a framework for applying agentic LLMs in code deobfuscation problems.

2 METHODOLOGY

2.1 Framework design

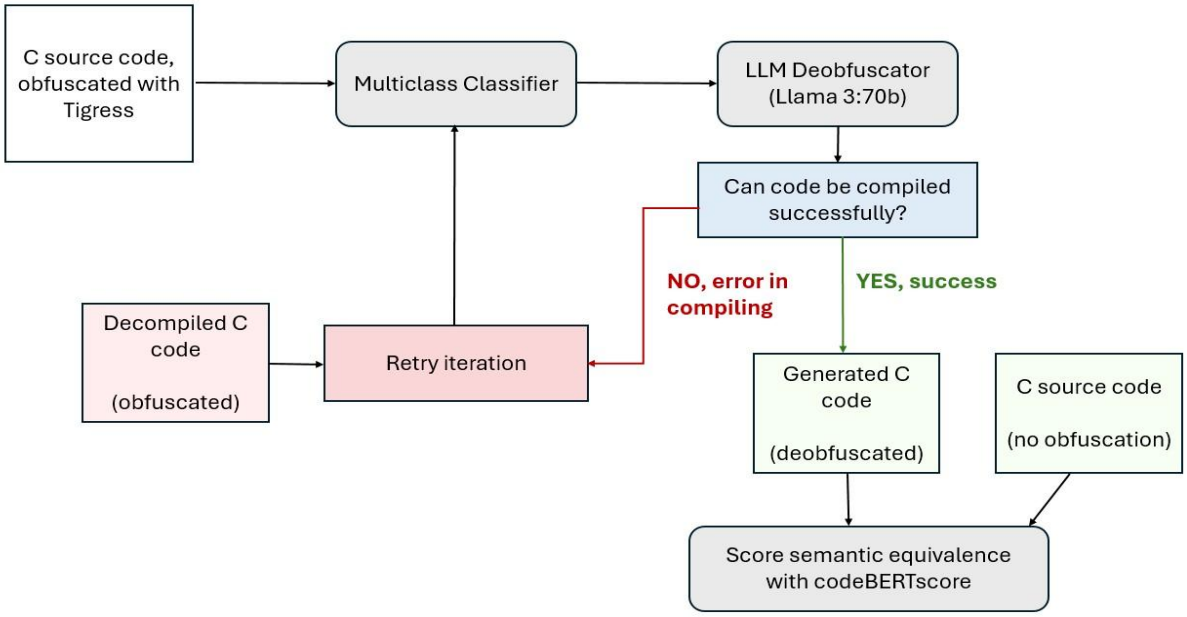

Fig 2.0 ALFREDO framework pipeline.

Agentic AI system. We implement an agentic AI system with LangGraph [13], defining three core steps: the *multiclass classifier, deobfuscator*, and *scorer*. Memory is maintained across steps, and tools such as Ghidra's Headless Analyzer aid the model in the deobfuscation process. The agentic system is extensible, allowing for the future addition of more sophisticated tools.

Multiclass classifier approach. The multiclass classifier tasks the Llama 3 model to identify a possible obfuscating transformation applied to the given code, with the aim of giving a clear direction for the deobfuscator. The classifier chooses from 4 named transformations (Flattening, AddOpaque, EncodeBranches, Virtualization) and produces a structured label which is passed along with the obfuscated code and system prompt to subsequent steps.

Deobfuscator. The obfuscating transformation, obfuscated code, and system prompt are then provided to a second instance of the Llama 3 model for the deobfuscation task. Given the often complex and deliberately convoluted nature of obfuscated code, we selected a model temperature of 0.4 to produce focused results while introducing a degree of randomness, encouraging the model to explore alternative solutions to deobfuscation problems over repetitive, inaccurate answers. Prompts were engineered and selected for highest model compliance to the instructions.

> {code}. This code has undergone {Flattening/AddOpaque/EncodeBranches/Virtualization} transformation. Your job is to deobfuscate this code. It should have the same functionality and semantics, but be deobfuscated. Only provide C code. Do not explain, and no additional notes, comments, or sentences. Do not add `. Your entire answer must be code, with no explanation. Your deobfuscated answer should compile without any errors.

Fig 2.1 Final prompt provided to Llama 3 model for the deobfuscation task

Iterative process. We observed that code generated by the model frequently produced compilation errors, rendering code ineffective. We improve model performance significantly by implementing an iterative process, wherein if generated code is not successfully compiled with GNU compiler, we repeat the process from the classifier step. We further supplement the model with decompiled versions of the obfuscated program, obtained by the agentic system through Headless Analysis, as we found that LLMs benefit from having pseudocode as a structured guide. [14] Additionally, even with obfuscated elements, decompiled pseudo C represents the code's original control flow with greater clarity and structure, improving deobfuscation accuracy. Should the compilation still fail after 4 iterative attempts, we abort the process, calculate the F-measure, and move on to the next file.

2.2 Experimental Setup

LLM model. We opt to evaluate our approach primarily using Meta Llama 3 70B [3] due to resource constraints. Meta Llama 3 70B performed comparable to GPT-3.5 and GPT-4.0. While its accuracy is lacking in multiple areas, the model is open-source and applicable to fine-tuning.

Dataset. Our curated dataset originates from the POJ-104 dataset [4] containing 32,000 solutions for 62 programming problems. We removed all files that failed to compile and obfuscated the remaining using the Tigress obfuscator [5] by applying four different transformations: Flattening, Virtualize, AddOpaque, and EncodeBranches, resulting in four obfuscated variants of each program alongside the original program with no obfuscations applied. We removed the programs that failed to be obfuscated for any of the four transformations and obtained 29,998 programs per transformation type, amounting to 119,992 obfuscated programs. We also used the Obfuscation Benchmarks dataset [11] with 99 flattened C programs for comparison with other deobfuscators.

Dataset	Flatten	Virtualize	AddOpaque	Encode Branches	Total
POJ-104	29998	29998	29998	29998	119992
Obfuscation Benchmarks	99	-	-	-	99

Table 2.2 Datasets used to evaluate ALFREDO.

Benchmarking. Among the obfuscating transformations chosen, there is a limited amount of existing work. EncodeBranches is a relatively unexplored obfuscation, while only one commercial tool, Binary Ninja [6], exists for automatic deobfuscation of the AddOpaque transformation. Currently, there are also no reliable automatic tools for deobfuscating Virtualization-protected code [7]. Most well-researched is control flow flattening (CFF), with several state-of-the-art tools available for automatic deobfuscation of flattened code such as MODeflattener [8], deflat [9], and CaDeCFF, whose source code is not publicly available. [10] Given these findings, we benchmarked our model's deobfuscation abilities against the two CFF deobfuscators, MODeflattener and deflat.

Evaluation metrics. We use codeBERTscore to calculate F-measure, a widely-used metric based on the harmonic mean of precision and recall, derived from token-level semantic comparisons

between the original C code and code generated by the deobfuscators. The score is a numerical value between 0 and 1, with higher values indicating higher semantic equivalence [12].

3 RESULTS

To evaluate various aspects of ALFREDO as a framework, we devised three research questions.
RQ1: How accurate is ALFREDO in recovering the original deobfuscated C code?
RQ2: How does specific obfuscating transformation applied affect ALFREDO's effectiveness?
RQ3: How does the multiclass classifier contribute to ALFREDO's deobfuscation ability?
In addition to results, we discuss the implications of our findings in Section 5.

3.1 RQ1: Model Performance

From testing ALFREDO on 1,500 obfuscated files in the modified POJ-104 dataset, we obtained an average F-measure of 0.77, indicating high semantic equivalence of the generated deobfuscated code to the original source code. A similar F-measure for ALFREDO was obtained when testing on the benchmark dataset, used to compare performance with MODeflattener and deflat. Because MODeflattener and deflat fail to generate deobfuscated code for programs in the POJ-104 dataset obfuscated with AddOpaque, EncodeBranches and Virtualization, we represented their F-measure as –.

		MODeflattener	**deflat**	**ALFREDO**
Average F-measure	Flattening	0.93	0.95	0.80
	AddOpaque	–	–	0.78
	EncodeBranches	–	–	0.84
	Virtualization	–	–	0.66
	Overall	–	–	0.77
Typical runtime (seconds)		0.09	3.43	9.42 ~ 52.8

Table 4.0 F-measure and runtime comparisons between the deobfuscators.

Although ALFREDO's accuracy already fulfills a high standard, it falls short of methodical deobfuscators like MODeflattener and deflat, which employ fixed techniques for analysis to produce highly accurate deobfuscations. ALFREDO also averages a longer typical runtime: the runtime for a deobfuscation attempt completed successfully on the first instance is 9.42 seconds on average, while maximum runtime for deobfuscating a program was 125.62 seconds, which typically indicated repeated failures in compilation of generated code. ALFREDO's runtime can largely be attributed to the nature of LangGraph as well as the iterative process necessitated by the nature of code generated by LLMs.

3.2 RQ2: Impact of specific obfuscating transformations

Obfuscating transformations vary in complexity, presenting differing degrees of challenge to deobfuscate. We determine if this distinction similarly applies to an LLM-based approach by comparing the deobfuscation success rates of ALFREDO for various transformations. In Table

4.0, Virtualization-protected code was most difficult to deobfuscate, with the lowest F-measure of 0.66, whereas deobfuscation accuracy for Flattening, AddOpaque, and EncodeBranches were all significantly higher, with F-measures within the range of 0.77 to 0.84. These findings align with existing knowledge that Virtualization is challenging to deobfuscate even by manual approaches. [15]

3.3 RQ3: Classifier effectiveness

Finally, we consider the effectiveness of the multiclass classifier. The rate of correct classifications was 29.01%. When testing ALFREDO with the absence of the multiclass classifier, we found that the average F-measure remained the same at 0.77, showing no apparent decline in performance. We hypothesise that other techniques such as Support Vector Machines and Random Forest are preferable to LLMs for classifying obfuscating transformations, and can increase the F-measure if integrated into the framework.

4 CONCLUSION

4.1 Discussion of results

Although ALFREDO's performance is lower than methodical deobfuscators like MODeflattener and deflat, we demonstrate how it is able to generate deobfuscated code reliably, with a high average semantic equivalence of 77.9% to the original. Additionally, ALFREDO is able to deobfuscate various transformations, while existing tools for automatic deobfuscation typically only support one transformation type, such as CFF. Lastly, due to its versatility, ALFREDO can be an intermediate solution where there is a lack of methodical deobfuscators in the first place.

However, ALFREDO's performance deteriorates for more challenging obfuscations like Virtualization. This suggests limitations to an LLM-based framework's capacity to deobfuscate code. Hence, we suggest ALFREDO to be supplemented with additional analysis tools such as control flow graphs or symbolic execution to fully realise its potential in code deobfuscation contexts.

4.2 Implications and future work

Societal impact. We believe that ALFREDO could be a viable tool for malware analysis and general analysis of obfuscated programs, able to deobfuscate code of a large range of transformations. Due to their rigid analysis methods, traditional deobfuscators may perform poorly in real-world contexts, where malware is often obfuscated with more than one transformation and deliberately designed to hinder methodical deobfuscation.

LLM-based frameworks like ALFREDO can mitigate this due to their ability to adapt and analyse obfuscated code. Additionally, agentic systems allow support with strategic analysis tools like taint analysis, maximising technical competence while retaining a flexible, creative decision-making approach. Finally, our work also demonstrates the strong potential of agentic LLM applications in an entirely new field, being to the best of our knowledge, the first existing work to investigate the usage of agentic LLMs in code deobfuscation and achieving a high accuracy rate of 77.9%.

Future work. Future research could test the integration of more sophisticated analysis tools with agentic LLM-based frameworks like ALFREDO. Given the diversity of such tools, it is important to determine which can be successfully combined with LLM capabilities to improve the accuracy and efficiency of deobfuscation. Exploring other machine learning techniques for the classifier and fine tuning the deobfuscator for specific transformations can further improve ALFREDO's performance as well. The effect of layering multiple obfuscating transformations on LLM deobfuscation ability was also outside the scope of this work, but could be an interesting area for future research, and simulate obfuscation in more realistic contexts.

ACKNOWLEDGEMENTS

We would like to thank our mentors Dr Siow Jing Kai and Ms Poh Hui-Li Phyllis for their unwavering support and guidance throughout this project. Additionally, we would also like to thank DSO National Laboratories for this research opportunity and for providing us with the resources for our project.

References

[1] Constantinos Patsakis, Fran Casino, Nikolaos Lykousas, Assessing LLMs in malicious code deobfuscation of real-world malware campaigns, https://doi.org/10.1016/j.eswa.2024.124912.

[2] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2024. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. ACM Trans. Inf. Syst. Just Accepted (November 2024). https://doi.org/10.1145/3703155

[3] Meta, "meta-llama/Meta-Llama-3-70B · Hugging Face," huggingface.co, Apr. 18, 2024. Available: https://huggingface.co/meta-llama/Meta-Llama-3-70B/tree/main. Accessed: Dec. 3, 2024.

[4] CodeXGLUE -- Clone Detection (POJ-104). Available: https://github.com/microsoft/CodeXGLUE/blob/main/Code-Code/Clone-detection-POJ-104/README.md. Accessed: Nov. 20, 2024.

[5] the tigress c obfuscator. Available: https://tigress.wtf/. Accessed: Nov. 19, 2024.

[6] Opaque Predicate Patcher (v1.1.6). Available: https://github.com/Vector35/OpaquePredicatePatcher/tree/master. Accessed: Dec. 20, 2024.

[7] Kochberger, P., Schrittwieser, S., Schweighofer, S., Kieseberg, P., & Weippl, E. (2021). SOK: Automatic Deobfuscation of Virtualization-protected Applications. Proceedings of the 17th International Conference on Availability, Reliability and Security, 1–15. https://doi.org/10.1145/3465481.3465772

[8] MODeflattener - Miasm's OLLVM Deflattener. Available: https://github.com/mrT4ntr4/MODeflattener. Accessed: Dec. 16, 2024.

[9] deflat. Available: https://github.com/pcy190/deflat. Accessed: Dec. 20, 2024.

[10] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. 2022. CaDeCFF: Compiler-Agnostic Deobfuscator of Control Flow Flattening. In Proceedings of the 13th Asia-Pacific Symposium on Internetware (Internetware '22). Association for Computing Machinery, New York, NY, USA, 282–291. https://doi.org/10.1145/3545258.3545269

[11] Obfuscation Benchmarks. Available: https://github.com/tum-i4/obfuscation-benchmarks. Accessed: Nov. 19, 2024.

[12] Zhou, S., Alon, U., Agarwal, S., & Neubig, G. (2023). Codebertscore: Evaluating code generation with pretrained models of code. arXiv preprint arXiv:2302.05527.

[13] LangGraph. Available: https://www.langchain.com/langgraph. Accessed: Dec. 15, 2024.

[14] Chae, H., Kim, Y., Kim, S., Ong, K. T., Kwak, B., Kim, M., Kim, S., Kwon, T., Chung, J., Yu, Y., & Yeo, J. (2024). Language Models as Compilers: Simulating Pseudocode Execution Improves Algorithmic Reasoning in Language Models. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2404.02575.

[15] Coogan, K., Lu, Gen., Debray, S. (2011). Deobfuscation of virtualization-obfuscated software: a semantics-based approach. CCS'11: the ACM Conference on Computer and Communications Security. https://doi.org/10.1145/2046707.2046739

[16] Miasm. Available: https://github.com/cea-sec/miasm. Accessed: Dec. 16, 2024.

[17] Ghidra Software Reverse Engineering Framework. Available: https://github.com/NationalSecurityAgency/ghidra. Accessed: Nov. 19, 2024.

[18] Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2024). A Survey on Large Language Models for Code Generation. arXiv preprint arXiv:2406.00515.

APPENDIX

Appendix A

An example illustrating transformations to code

(i) Original sample of C code in POJ-104 dataset, depicting a standard program to find the length of the Longest Increasing Subsequence (LIS). Labelled as 10_57.c, with 28 lines of code.

```
void main()
{
    int a[50]={0},flag[50],max,i,j,count=1,st=0;
    int num;
    scanf("%d",&num);
    for (i=0;i<num;i++)
    {
        scanf("%d",&a[i]);
        flag[i]=1;
    }
    for (i=0;i<num;i++)
    {
        for(j=0;j<i;j ++)
        {
            if(a[j]>=a[i])
            {
                flag[i]=flag[j]+1>flag[i]?flag[j]+1:flag[i];
            }
        }
    }
    max = flag[0];
    for (i=1;i<num;i++)
    {
        if (flag[i]>max)
        max=flag[i];
    }
    printf("%d\n",max);
}
```

(ii) Obfuscated C code, depicting the same program but with the Encode Branches transformation applied. Labelled as 10_57_InitBranchFuns.c, with 2,058 lines of code.

```
struct _IO_codecvt ;
struct _IO_FILE ;
struct _IO_marker ;
struct _IO_wide_data ;
union __anonunion_pthread_rwlockattr_t_145707745 ;
struct __anonstruct___mbstate_t_996288157 ;
```

```
struct __anonstruct___g1_start32_961093919 ;
struct drand48_data ;
struct __anonstruct___wseq32_961093918 ;
union __anonunion_pthread_rwlock_t_656928968 ;
union __anonunion____missing_field_name_744779506 ;
struct __anonstruct___fsid_t_109580352 ;
enum __anonenum_idtype_t_558242672 ;
struct __pthread_mutex_s ;
struct __pthread_rwlock_arch_t ;
   char *__builtin_strchr(char * , int  ) ;  */
typedef unsigned long __u_quad_t;
typedef unsigned int __u_int;
typedef __u_int u_int;
typedef unsigned int __mode_t;
extern int putchar_unlocked(int __c ) ;
   int __builtin_ctz(unsigned int  ) ;  */
extern  __attribute__((__nothrow__)) long ( __attribute__((__leaf__))
mrand48)(void) ;
typedef unsigned long __ino_t;
typedef __ino_t ino_t;
   float __builtin_acosf(float  ) ;  */
typedef struct _IO_FILE FILE;
extern int putc(int __c , FILE *__stream ) ;
typedef long __blksize_t;
typedef __blksize_t blksize_t;
   int __builtin_strcmp(char const   * , char const   * ) ;  */
   double __builtin_sinh(double  ) ;  */
typedef int __daddr_t;
typedef __daddr_t daddr_t;
typedef unsigned long size_t;
extern size_t fread_unlocked(void * __restrict  __ptr , size_t __size ,
size_t __n ,
                             FILE * __restrict  __stream ) ;
   float __builtin_fabsf(float  ) ;  */
typedef unsigned long __fsfilcnt_t;
typedef __fsfilcnt_t fsfilcnt_t;
extern  __attribute__((__nothrow__)) char *( __attribute__((__leaf__))
ctermid)(char *__s ) ;
   float __builtin_fmodf(float  ) ;  */
   void __builtin_va_arg(__builtin_va_list  , unsigned long  , void * ) ;
*/
extern int fputs(char const   * __restrict  __s , FILE * __restrict
__stream ) ;
extern  __attribute__((__nothrow__)) void ( __attribute__((__leaf__))
funlockfile)(FILE *__stream ) ;
typedef unsigned long __nlink_t;
typedef __nlink_t nlink_t;
...
```

```
...
```

(iii) C code, deobfuscated with ALFREDO to resemble 10_57.c, with 29 lines of code. The obfuscated code from (ii) was provided to ALFREDO, and the deobfuscated code shown below was returned as output, with an F-measure of 0.90 when compared to the original C code in (i).

```
int main() {
    int a[N] = {0}, num, i, j;
    int flag[N] = {0};
    int max;
    scanf("%d", &num);
    for (i = 0; i < num; i++) {
       scanf("%d", &a[i]);
       flag[i] = 1;
    }
    for (i = 0; i < num; i++) {
        for (j = 0; j < i; j++) {
           if (a[j] >= a[i]) {
              if (flag[j] + 1 > flag[i])
                 flag[i] = flag[j] + 1;
              else
                 flag[i] = flag[i];
              }
           }
        }
       max = flag[0];
       for (i = 1; i < num; i++) {
          if (flag[i] > max)
          max = flag[i];
       }
    printf("%d\n", max);
    return 0;
}
```