JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 30 No. 1, pp. 73-85, January 2025
https://doi.org/10.9708/jksci.2025.30.01.073

Design of an Automated Framework for Applying Generative AI-Based Source Code Obfuscation Techniques

Jihun Han*, Seung-A Park**, Joonseo Ha**, Chang-min Lee*, Kyung-mi Jung***,
Mee Lan Han****, Jun-Seob Kim****, Geumhwan Cho****

*Student, Dept. of AI Cyber Security, Korea University, Sejong, Korea
**Master's Course Student, Dept. of Cyber Security, Korea University, Sejong, Korea
***Researcher, Edge Cloud Data Security Research Center, Korea University, Sejong, Korea
****Professor, Dept. of AI Cyber Security, Korea University, Sejong, Korea

[Abstract]

Source code obfuscation is an essential technique for software security and intellectual property protection. Traditional source code obfuscation methods depend on human-driven processes or predefined algorithms implemented by obfuscation tools. As a result, it becomes difficult to effectively manage the quality and complexity of obfuscated code. However, given that the required level of obfuscation differs based on the threat model, we need to develop techniques that can flexibly adjust the level and complexity of obfuscation. This paper proposes an automated framework that generates obfuscated source code from original source code using GPT-4o. The proposed framework generates prompts based on 12 obfuscation techniques and utilizes these prompts as input to GPT-4o. The generated obfuscated source code is executed and verified to ensure that it maintains the same intended functionality as the original code. Experimental results demonstrate that the proposed framework successfully executed 48 out of 60 obfuscated source codes while effectively applying the intended obfuscation techniques.

▸ **Key words**: Generative Artificial Intelligence, Large Language Model(LLM), Source Code Generation, Source Code Obfuscation, Obfuscation Technique, LLM Performance Evaluation

• First Author: Jihun Han, Corresponding Author: Geumhwan Cho
*Jihun Han (smallstone00@korea.ac.kr), Dept. of AI Cyber Security, Korea University
**Seung-A Park (hn8909@korea.ac.kr), Dept. of Cyber Security, Korea University
**Joonseo Ha (cujs1106@korea.ac.kr), Dept. of Cyber Security, Korea University
*Chang-min Lee (2020270127@korea.ac.kr), Dept. of AI Cyber Security, Korea University
***Kyung-mi Jung (kmjung@korea.ac.kr), Edge Cloud Data Security Research Center, Korea University
****Mee Lan Han (blosst@korea.ac.kr), Dept. of AI Cyber Security, Korea University
****Jun-Seob Kim (jskim90@korea.ac.kr), Dept. of AI Cyber Security, Korea University
****Geumhwan Cho (geumhwan@korea.ac.kr), Dept. of AI Cyber Security, Korea University
• Received: 2024. 12. 09, Revised: 2024. 12. 26, Accepted: 2025. 01. 02.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

[요 약]

소스 코드 난독화는 소프트웨어 보안과 지적 재산 보호를 위해 사용되는 중요한 기술이다. 기존의 소스 코드 난독화 작업은 수작업이나 난독화 도구에 의한 고정된 알고리즘에 의존해 유연성이 부족하며 난독화된 코드의 품질과 복잡성을 효과적으로 제어하기 어렵다. 그러나 위협 모델에 따라 요구되는 난독화 수준이 달라지므로 다양한 난독화 수준과 복잡성을 유연하게 조정할 수 있는 기술이 필요하다. 본 논문에서는 GPT-4o를 활용하여 원본 소스 코드를 난독화된 소스 코드로 자동 생성하는 프레임워크를 제안한다. 제안하는 프레임워크는 12개의 난독화 기법을 기반으로 프롬프트를 생성해 GPT-4o에 입력으로 사용한다. 생성된 난독화 소스 코드의 실행 결과를 검증하여 원본과 동일한 의도를 유지하는지 확인한다. 실험 결과, 제안된 프레임워크는 난독화된 소스 코드 60개 중 48개가 원본과 동일하게 의도한 대로 성공적으로 실행되었으며 난독화 기법도 제대로 적용된 것을 확인하였다.

▸ **주제어**: 생성형 인공지능, 대규모 언어 모델, 소스 코드 생성, 소스 코드 난독화, 소스 코드 난독화 기법, 대규모 언어 모델 성능 평가

I. Introduction

소스 코드 난독화(source code obfuscation)는 프로그래밍 언어로 작성된 코드의 일부 또는 전체를 변형하여 소스 코드를 이해하기 어렵게 만드는 기법으로써, 소프트웨어 보안과 지적재산권 보호를 위해 사용되는 중요한 기술이다. 공격자들은 개인정보를 처리하는 애플리케이션과 같은 특정 프로그램을 해킹하여 프로그램을 조작한 다음, 프로그램의 조작 여부를 인지하지 못한 상태로 프로그램을 실행하도록 공격을 수행한다[1]. 소스 코드 난독화는 공격자들이 역공학을 통해 프로그램을 쉽게 조작하는 것을 방지하는 중요한 역할을 한다. 또한 소스 코드 내의 핵심적인 아이디어나 알고리즘이 불법으로 복제되는 것을 방지함으로써 지적재산권을 보호하기도 하고 소프트웨어의 취약점을 탐지하거나 취약점을 이용한 악성코드 제작을 사전에 방지하는 역할을 한다.

그러나 기존의 소스 코드 난독화 작업은 수작업이나 난독화 도구에 의한 고정된 알고리즘에 의존해 유연성이 부족하며 난독화된 코드의 품질과 복잡성을 효과적으로 제어하기 어렵다. 또한 난독화된 소스 코드와 알고리즘의 계산복잡도로 인해 실행 과정에서 불필요한 자원 소모가 수반되기 때문에 난독화 수준과 복잡성을 유연하게 조정할 수 있는 기술이 필요하다[2].

이에, 본 논문은 생성형 AI를 기반으로 Python 스크립트에 소스 코드 난독화 기법을 적용하여 새로운 스크립트 코드를 생성하는 자동화 프레임워크를 설계하는 것을 목적으로 한다. 현재까지의 연구를 살펴본 결과, 생성형 AI의 fine-tuning 없이 프롬프트 엔지니어링만을 이용한 소스 코드 난독화 연구는 진행된 바 없다. 본 논문은 소스 코드 난독화 기법의 자동화 적용을 통해 코드 개발자들의 작업 시간 단축과 선별적 난독화 기법 적용의 편의성 향상을 제공한다. 본 문의 기여도를 정리하면 다음과 같다.

- 본 연구는 생성형 AI를 기반으로 소스 코드 난독화 과정을 자동화하는 프레임워크를 제시하였다.
- 실용적인 소스 코드 생성을 위해 폭 넓은 자료 조사를 바탕으로 프롬프트 엔지니어링을 수행함으로써 소스 코드 난독화 기법 적용에 효과적인 프롬프트를 설계하였다.
- 다방면의 평가 지표를 바탕으로 프레임워크에 대한 세부적인 평가를 수행하였으며, 생성형 AI를 활용한 소스 코드 난독화 적용 기법의 가능성을 확인하였다.

본 논문의 구성은 다음과 같다. II 장 배경지식과 참고문헌을 서술하고, III 장에서는 본 논문에서 제안하는 전체 프레임워크와 몇 가지 소스 코드 난독화 기술을 설명한다. 그리고 IV 장은 실험을 통해 소스 코드 난독화 기법 적용 자동화 프레임워크의 동작을 검증하고 성능을 평가한 결과를 제시한다. 마지막으로 V 장에서 결론 및 추후 연구 방향을 서술하며 마무리한다.

II. Backgrounds and Related Works

1. Backgrounds

1.1 Generative AI and LLM

언어 모델에서 진화한 대형 언어 모델(Large Language Model, LLM)은 방대한 데이터 세트로부터 광범위한 훈련을 거쳐 인간의 언어를 밀접하게 모방하는 방식으로 텍스트를 이해하고 생성한다. 또한, LLM 모델은 방대한 텍스트 데이터 전처리를 통해 다듬어진 수많은 파라미터(보통 수십억 웨이트 이상)를 부여받은 인공지능 신경망이다[3]. LLM 모델의 사용은 Web-UI 방식과 API 키 발급을 통해 필요한 파이썬 패키지 다운로드와 라이브러리 임포트(import)할 수 있다.

LLM 모델은 주로 지속적으로 발생하는 언어 텍스트를 처리하고 생성하는 데 특화되어 있다 보니, 다음과 같이 LLM 모델이 최소한 가져야 하는 몇 가지 핵심 기능이 존재한다[4].

- 자연어 텍스트에 대한 심층적인 이해와 해석을 바탕으로 번역과 같은 다양한 언어 관련 작업을 수행할 수 있어야 함
- 프롬프트가 표시되면 문장 완성이라던가 문단 구성과 같은 인간과 유사한 텍스트를 생성할 수 있는 능력이 있어야 함
- 도메인 전문성과 같은 요소를 고려하여 맥락적 인식을 보여야 함
- 문제 해결 및 의사 결정에 탁월해야 함

위와 같은 핵심 기능을 바탕으로 LLM 모델은 인간의 언어 형태인 입력 프롬프트를 기반으로 콘텐츠를 생성하는 생성형 AI에 사용될 수 있다. 생성형 인공지능(Generative artificial intelligence) 또는 생성형 AI(Generative AI)는 데이터의 패턴을 기반으로 텍스트나 이미지와 같은 새로운 콘텐츠를 만들 수 있는 AI 모델의 한 유형이다[5]. 생성형 AI는 단순히 행동을 판별하거나 분류 또는 회귀 문제를 해결하는 방식인 예측적 기계 학습 시스템과 달리, 입력 정보를 잠재적인 고차원 공간에 매핑할 수 있는 말뭉치(Corpus, 코퍼스)를 통해 텍스트, 이미지, 음성과 같은 콘텐츠 생성을 위한 확률적 행동을 자율적으로 수행할 수 있는 모델을 포함한다 [6].

생성형 AI 중에서 입력 텍스트에 대해 출력도 텍스트로 도출되는 모델(Text-to-Text models)에는 대표적으로 ChatGPT, Claude, Gemini, LLaMA 등이 있다. ChatGPT는 개방형 대화에 탁월하고, LLaMA와 Gemini는 전문 영역에 대한 지원에 중점을 두고 있다. LLaMA는 오픈소스로써 공개되고 있으며, 파인튜닝 등에 활용되는 반면, Gemini는 답변 시 입력한 지시와 관련한 내용을 구글에서 검색하여 최신 지식을 context에 추가하는 방식이다[7].

Table 1. Categories for generative AI models [7].

Category	Models
Text-to-image models	■ DALL·E 2 ■ IMAGEN ■ Muse
Text-to-3D models	■ Dreamfusion ■ Magic3D
Image-to-Text models	■ Flamingo ■ VisualGPT
Text-to-Video models	■ Phenaki ■ Soundify
Text-to-Audio models	■ AudioLM ■ Jukebox ■ Whisper
Text-to-Text models	■ ChatGPT ■ LLaMA ■ Gemini
Text-to-Code models	■ Codex ■ Alphacode
Text-to-Science models	■ Galactica ■ Minerva

1.2 Prompt Engineering

프롬프트 엔지니어링은 생성형 AI 시스템이 고품질의 결과를 생성하도록 입력을 위한 프롬프트를 작성, 정제 및 최적화하는 프로세스를 말한다. 이러한 프로세스는 LLM 모델과 상호작용하면서 LLM 모델의 잠재력을 최대한 활용하며, 분석가가 적용하고자 하는 도메인에 대해 보다 쉽게 접근 가능하도록 지원하는 중요한 역할을 한다[8]. 프롬프트 엔지니어링의 중요성은 모델 응답을 얼마나 정확하게 신속하게 안내하는지에 따라 강조된다. 이를 통해, LLM 모델의 다양성과 관련성은 더욱 강화되고, 여러 도메인으로 확장할 수 있다[9, 10].

프롬프트는 LLM 모델로부터 출력을 생성하기 위해 입력하는 텍스트를 말한다. 응답 정확성을 높이고 모호성은 줄이는 방향으로 잘 설계된 프롬프트는 LLM 모델이 정보를 처리하는 과정에서 발생하는 대표적인 오류인 기계 환각 (Hallucination), 즉, 모순된 문장, 프롬프트와의 모순, 사실 왜곡, 허위 정보 생성과 같은 문제에 적절하게 대응할 수 있다[11]. 다만, LLM 모델은 학습 데이터에 기반하므로 프롬프트에 대한 응답 역시 학습 데이터의 품질과 편견이 반영될 수 있고, 프롬프트를 지나치게 구체적으로 서술하거나 특정한 방식으로 최적화를 진행하다 보면, 과적합이 발생할 위험이 존재한다.

1.3 Source Code Obfuscation

소스 코드 난독화는 소스 코드를 의도적으로 복잡하게 만들어 해석을 어렵게 만드는 기술이다[12]. 소프트웨어를 보호하는 긍정적인 용도로 사용될 수 있지만, 악성코드 개발자가 이를 악용하여 악성코드 분석을 어렵게 만드는 데에도 활용될 수 있다. 일반적인 소스 코드 난독화 기법으로는 Control Flow Obfuscation[13], Data Obfuscation[12], Instruction Obfuscation[14] 등이 주로 사용되며, 지속적으로 새로운 소스 코드 난독화 기법들이 개발되고 있다. 이에 따라 소스 코드 난독화 기법을 탐지하고 분석하기 위한 기술도 함께 발전하고 있다.

정적 분석과 동적 분석은 소스 코드 난독화를 탐지하기 위한 주요 방법으로 사용된다. 정적 분석은 난독화된 코드를 직접 분석하여 패턴을 추출하는 방식으로 역공학 도구와 디스어셈블러가 주로 사용된다. 반면, 동적 분석은 실행 시점에서 코드의 동작을 모니터링하여 소스 코드 난독화된 부분을 복구하거나 탐지하는 방식으로 샌드박스 환경과 메모리 디버깅 도구가 활용될 수 있다[12].

2. Related Works

본 절에서는 소스 코드 생성의 자동화와 관련된 연구 및 이를 지원하는 사전 학습 모델과 오픈 소스 난독화 도구에 대해 다룬다.

2.1 Automated Source Code Generation

소프트웨어 개발의 효율성을 높이기 위해 자연어 설명으로부터 소스 코드를 생성하는 기술의 연구가 활발히 이루어지고 있다[15]. 이러한 수요에 부응하기 위해, 최근 LLM은 데이터 수집, 학습 기술, 성능 평가, 실제 응용 등의 목표를 가지고 개발되고 있으며, 여러 벤치마크[16, 17, 18]를 통해 LLM의 코드 생성 능력이 지속적으로 개선되고 있다.

대표적인 연구 사례로 CodeBERT가 있다[19]. Feng 등은 CodeBERT를 통해 프로그래밍 언어와 자연어를 모두 이해할 수 있는 모델을 제안하였으며, 이를 활용해 코드 검색, 코드 완성, 버그 수정 등의 작업에서 뛰어난 성능을 보였다. 이와 유사하게 Wang 등[20]의 CodeT5는 소스 코드의 식별자 인식을 강화하여 코드 이해와 생성을 효과적으로 수행할 수 있도록 설계된 사전 학습 모델이다.

2021년, OpenAI는 자연어 명령어로 소스 코드를 생성할 수 있는 Codex[21]를 발표하였으며, 이를 기반으로 GitHub Copilot과 같은 도구에서 반복적인 작업을 자동화하는 데 기여하였다. 또한, Magicoder[22]는 오픈소스 코드 스니펫을 활용해 다양한 코딩 문제를 학습하고, 모델의 편향성을 줄이며 현실적인 데이터를 제공함으로써 HumanEval 벤치마크에서 우수한 성능을 보여주었다. 이 외에도 Meta의 TestGen-LLM[23]은 단위 테스트 생성을 자동화하여 테스트 커버리지와 안정성을 개선하였으며, Huang 등[24]은 코드 실행 효율성을 높이기 위해 프로파일 데이터를 기반으로 최적화하는 기술인 SOAP를 제안하였다.

Hu 등[25]은 기존의 고정된 알고리즘에 기반하는 디컴파일러의 한계를 극복하기 위해 생성형 AI 기반 디컴파일러인 DeGPT를 제시하였다. 이는 의미 있는 변수명과 주석을 통해, 생성된 고급 언어 소스 코드의 가독성을 향상시킬 뿐만 아니라, 생성된 코드의 구조를 최적화할 수 있다는 점에서 강점이 있다. 또한, DeGPT가 생성한 소스 코드는 룰 기반 디컴파일러 대비 평균 75.6%의 구조적 단순성을 보였으며, 99.2%의 비율로 올바른 주석을 생성하였다. 이는 생성형 AI의 고급 언어 이해 및 소스 코드 생성 능력을 보여준다는 점에서 큰 의미를 갖는다.

이러한 연구들은 소스 코드 생성 자동화를 위한 LLM의 잠재력을 입증하며, 소프트웨어 개발의 생산성과 정확성을 높이는 데 기여하고 있다. LLM은 코드 검색, 생성, 테스트 자동화 등 다양한 작업에서 효율성을 증대시켜 개발자들이 반복적인 작업에서 벗어날 수 있도록 돕는다. 이를 통해 LLM 기반 기술은 소프트웨어 개발의 중요한 도구로 자리 잡고 있으며, 그 활용 가능성은 점점 더 확대되고 있다.

2.2 Existing Source Code Obfuscation Tools

Pyminifier[27]는 주어진 Python 스크립트의 변수, 함수·메서드, 클래스 등에 대해 이름 맹글링(Name Mangling) 기반의 난독화를 수행하는 도구이다. 이 도구는 비교적 빠른 실행 시간을 제공하지만 클래스 속성에 대한 맹글링을 지원하지 않으며, f-string 형식의 문자열 출력 구문에서는 난독화를 수행하지 않는다.

Opy[28]는 이름 맹글링과 문자열 암호화를 수행하는 도구로 Pyminifier와 마찬가지로 빠른 실행 시간을 보인다. 그러나 이 도구 역시 f-string 형식의 문자열 출력 구문을 처리하지 못하며, 제어 흐름 난독화와 같은 고급 난독화 기법을 적용할 수 없다. 또한, 문자열 리터럴과 관련한 주석 처리 방법의 제한, '__'로 시작하는 메서드에 대한 이름 맹글링 미지원 등과 같은 한계가 존재한다.

본 논문에서는 난독화 성능을 평가하기 위해 두 가지 오픈소스 도구와 비교 분석을 수행하였다.

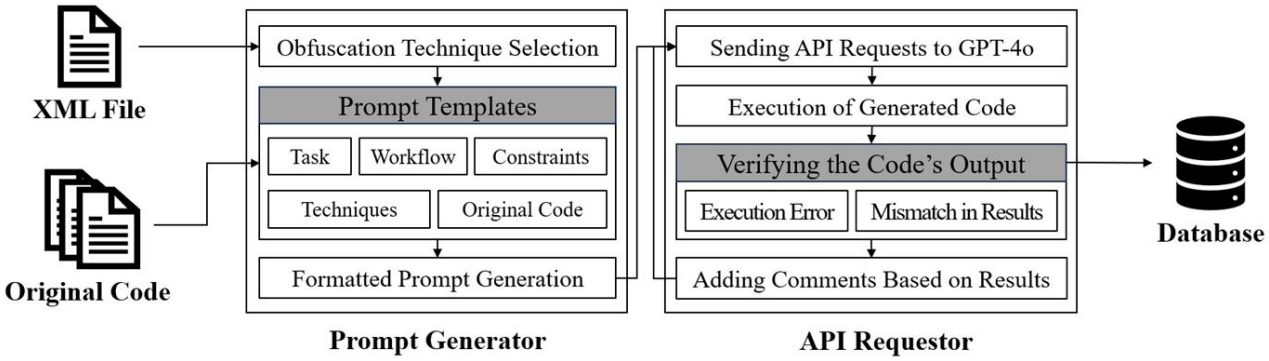

Fig. 1. Obfuscated code generation framework overview.

III. The Proposed Framework

본 장에서는 제시한 생성형 AI 기반의 소스 코드 난독화 기법 적용 자동화 프레임워크 설계 및 상세에 관해 논의한다.

1. Framework Overview

생성형 AI 기반 소스 코드 난독화 기법 적용 자동화 프레임워크는 소스 코드와 특정 소스 코드 난독화 기법이 입력으로 주어졌을 때, 형식에 맞춰 생성한 프롬프트를 바탕으로 API 요청을 수행함으로써 난독화된 코드를 얻는 일련의 과정을 자동화하는 프레임워크이다. Fig. 1은 본 연구에서 제시한 프레임워크를 나타내며, 크게 두 부분으로 나뉜다. 먼저, 프롬프트 생성기는 입력 소스 코드와 함께 작업 수행 내용, 순서, 제약 사항 등을 명시한 프롬프트를 생성한다. 반면, API 요청기는 앞서 생성한 프롬프트를 바탕으로 실제 소스 코드 난독화 기법이 적용된 소스 코드를 생성하도록 API 요청을 수행하며, 코드 실행 결과를 확인하고 경우에 따라 재요청을 통해 코드를 정정한 후, 데이터베이스에 저장한다.

Algorithm 1 Prompt Generator

```
Input: original codes (list of python codes),
       XML file (obfuscation techniques)
Output: prompts (list of prompt)
parse obfuscation techniques from XML file
generate obfuscation technique combinations T
for i to T length do
    for code in (original codes) do
        generate a prompt ← prompt templates, T[i], code
        save the prompt to DB
        save the amount of token for the prompt to DB
    end for
end for
```

Fig. 2. Pseudo code for prompt generator.

2. Prompt Engineering

프롬프트 생성기는 생성형 AI가 수행해야 하는 항목을 명시한 프롬프트의 목록을 자동으로 생성하는 도구이다. 해당 도구는 소스 코드 난독화 기법을 적용하고자 하는 원본 코드와 소스 코드 난독화 기법이 명시되어 있는 XML 파일을 입력으로 하여, 프롬프트를 생성한 뒤 이를 DB에 저장한다. Fig. 2는 프롬프트 생성기의 동작 과정에 관한 의사 코드이다.

프롬프트 생성기의 동작 과정은 다음과 같다. 먼저, 프롬프트 생성기는 입력받은 XML 파일의 내용을 파싱하여 소스 코드 난독화 기법을 저장하고, 이를 바탕으로 적용하고자 하는 소스 코드 난독화 기법의 조합을 생성한다. 이때 XML 파일은 파싱의 용이성을 위해 사용되었으며, 실제 프레임워크 구현의 동작 시에는 하나의 입력 소스 코드 당 하나의 소스 코드 난독화 기법을 적용하도록 하여 실험을 진행하였다. 이후, 소스 코드 난독화 기법 조합을 순회하며, 미리 작성한 프롬프트 템플릿과 선택된 소스 코드 난독화 기법, 원본 소스 코드를 입력으로 프롬프트를 생성한다. 마지막으로, 생성된 프롬프트는 데이터베이스에 저장되며, 대략적인 비용 계산을 위해 사용하고자 프롬프트의 토큰량을 계산하여 함께 저장한다.

사용하는 프롬프트는 각 목적과 수행 내용, 준수 사항 등을 명확하게 설명하고 있어야 한다. 생성형 AI는 동일한 프롬프트가 주어져도 상황에 따라 다른 결과를 반환하며 결과의 통제가 매우 어려운 문제가 있다. 이로 인해 프롬프트를 상세하고 명확히 제공함으로써 최대한 결과를 통제하기 위해 노력할 필요가 있다. 따라서, 본 연구에서 사용한 프롬프트는 각 항목을 구분하여 명시하고 불분명한 표현 대신 분명한 표현을 사용하고자 하였다.

프롬프트는 크게 다섯 가지의 구별된 task, workflow, constraints, techniques, original code 항목으로 구성되며, 프롬프트 생성기는 각 항목을 결합함으로써 최종적인 프롬프트를 생성한다. Fig. 3은 본 프레임워크에서 사용된 프롬프트를 나타낸 것이다. 먼저, Task 항목은 생성형 AI가 수행해야 하는 업무를 명시한 항목으로, 소스 코드 난독화 기법과 원본 소스 코드가 주어졌을 때, 원본 소

```
# Task:
You are provided with a specific code obfuscation technique, and the original script (original_script.py). Your objective is as follows:

Apply the provided obfuscation technique and its options to the original Python script (original_script.py).
Generate and obfuscated version of the original script.

# Workflow:
1. Apply the specified obfuscation technique to the original Python script (original_script.py).
3. Generate the obfuscated Python script.
4. Output the final obfuscated code.

# Constraints:
Note that, you must not explain the process, just focus on making the final obfuscated code.
The output must only contain the code generated as a result, without any explanation.

The examples provided are just examples, you must apply the given obfuscation techniques based on the (original_script.py).

Resulting code must be executable and contain no syntax errors.

# Obfuscation Technique:
  <technique name="Obfuscation">
    <description>Obfuscation Technique: A set of methods designed to make source code difficult to understand and analyze without altering the actual behavior or output of the program. These techniques aim to protect intellectual property, sensitive information, and code logic by obscuring code structure and increasing the complexity of reverse engineering.</description>
      <option name="Function Encoding">
        <description>Encode function definitions or calls using Base64 to hide their behavior from static analysis.</description>
        **Example:**
        <python>
            {{Example}}
        </python>
      </option>
    </combination_item>
  </technique>

# Original Code:
original_script.py - The following original Python script to be obfuscated and converted.
(original_script.py)
{{script_content}}
```

Fig. 3. Prompt example for generating obfuscated code: a ratio comparing the complexity of the obfuscated code to the original code, based on Halstead's Complexity Measures.

스 코드에 주어진 소스 코드 난독화 기법을 적용하고, 결과적으로 난독화된 소스 코드를 생성하도록 명시하고 있다. Workflow 항목은 생성형 AI가 수행할 작업의 순서를 명시하며, task 항목에 명시된 작업의 순서를 표시한다. Constraints 항목은 그 외 작업 수행에 있어서 지켜야 하는 사항을 나타내며, 본 프레임워크에서는 모델의 응답이 사용하는 토큰량을 줄이기 위해 부가적인 설명 없이 코드만을 요구하거나, 실행 가능하며 문법적 오류가 없어야 하고, 소스 코드 난독화 기법과 함께 제공되는 예시를 참고만 하도록 지시하였다. 나아가 technique에서는 선정한 소스 코드 난독화 기법을 XML 형식으로 제공하며, 이와 함께 짧은 예시를 제공한다. 이때 XML 형식은 복수의 소스 코드 난독화 기법 조합을 적용한 경우에 대한 프롬프트의 확장성을 고려하여 채택되었다. 마지막으로, original code에서는 원본 소스 코드의 내용을 포함한다.

3. API Requester

API 요청기는 생성형 AI 모델에 대한 API 요청을 수행하고 결과를 확인 및 저장하는 도구이다. 해당 도구는 앞서 생성한 프롬프트 목록을 입력으로 하여 API 요청을 통해 난독화된 코드를 생성한 뒤, DB에 저장한다. Fig. 4는 API 요청기의 동작 과정에 관한 의사 코드를 나타낸다.

Algorithm 2 API Requester

```
Input: prompts (list of prompts) p ∈ P,
       original codes
Output: obfuscated codes (list of codes)
for p in P do
    for i = 0 to max_retries do
        Request a response for p using GPT-4o
        Update DB with generated code and token count
        Execute the generated code
        if Error occurs during execution then
            Add comments to p with error information
        else
            Compare execution results with the original code
            if Results differ then
                Add comments to p with differences
            else
                break (Exit retry loop if successful)
            end if
        end if
    end for
end for
```

Fig. 4. Pseudo code for API requester.

API 요청기의 동작 과정은 다음과 같다. 먼저, 주어진 프롬프트를 바탕으로 생성형 AI 모델인 GPT-4o에 API 요청을 보낸다. 이후, 반환된 응답에서 코드 부분을 파싱한 뒤 이를 DB에 저장하고 반환받은 코드를 실행한다. 만약 코드 실행 중 오류가 발생할 경우, 발생한 오류 정보와 함께 새로운 코멘트를 추가하여 재요청을 수행한다. API를 통해 요청을 보낼 때, 모델이 이전 요청 내용을 기억하지 않으므로 최초 요청 프롬프트와 반환된 코드를 함께 포함하여 요청하도록 구성하였다.

또한, 오류가 발생하지 않을 경우엔 원본 코드의 실행

결과와 비교하여 각 소스 코드의 실행 결과가 동일한지 비교하며, 만약 서로 다른 결과를 출력한다면 각각의 결과를 프롬프트에 포함하여 재요청하도록 설계하였다. 마지막으로, 이를 최대 재요청 횟수까지 반복하도록 하여 난독화된 코드를 생성하도록 하였으며, 실제 실험에서 사용된 재요청 횟수는 5회이다.

프레임워크에서 사용된 데이터베이스는 Table 2와 같이 테이블을 구성하여 사용하였다. 테이블은 크게 11개의 열로 구성되어 있으며, 코드 생성 후 각 코드의 생성 과정 및 결과를 분석하기 위해 필요한 값을 저장하도록 하였다. 먼저, 테이블은 소스 코드 난독화를 위해 선택된 난독화 기법과 원본 코드명, 원본 코드를 저장할 수 있다. 이는 생성된 결과물이 어떤 원본 코드를 바탕으로 어떤 소스 코드 난독화 기법을 적용했는지 구분하기 위함이다. 다음은 코드 생성을 위해 사용된 프롬프트와 그에 해당하는 토큰량이 저장된다. 이는 앞선 프롬프트 생성기를 통해 제작된 프롬프트를 저장하는 항목으로, 난독화된 코드 생성에 어떤 프롬프트가 사용되었는지 확인할 수 있도록 하며, 토큰량은 API 요청을 위해 사용될 대략적인 비용을 추산할 수 있도록 돕는다. 또한, 테이블은 API 요청 이후 생성된 코드와 그에 따른 토큰량을 저장하도록 하였으며, 이는 본 연구에서 제시하는 프레임워크의 주요 결과물이다. 이때 생성형 AI의 API 요청 시 입력인 프롬프트와 응답인 결과물의 각 토큰량에 따라 청구되는 비용이 다르기에 앞선 프롬프트 토큰량과 구분하여 저장하도록 하였다. 마지막으로, 생성된 코드의 실행 결과에 따라 재요청을 수행하는 경우를 위해 재요청 횟수와 이때 사용된 최종적인 프롬프트, 최종적으로 반환 받은 전체 응답을 저장하도록 구성하였다.

Table 2. Table Components in database.

No.	Column Name	Data Type
1	id	INTEGER
2	obfuscation_technique	TEXT
3	original_code_name	TEXT
4	original_code	TEXT
5	prompt	TEXT
6	prompt_token	INTEGER
7	obfuscated_code	TEXT
8	obfuscated_code_token	INTEGER
9	retry_num	TEXT
10	final_prompt	TEXT
11	entire_response	TEXT

4. Obfuscation Techniques

본 논문에서는 총 12가지의 소스 코드 난독화 기법(Obfuscation Techniques)을 사용하여 소스 코드를 난독화한다. 소스 코드 난독화 기법은 난독화를 적용할 대상과, 난독화를 수행할 방식을 선택하여 사용하게 된다. 3가지 소스 코드 난독화 대상과 4가지 소스 코드 난독화 방식을 조합하여 총 12가지의 소스 코드 난독화 기법을 정의하였으며, 이를 정리하면 Table 3과 같다. 소스 코드 난독화가 적용될 대상에는 소스 코드를 구성하는 요소 중 변수, 함수, 객체가 해당된다. 이는 표에서 Obfuscation Component를 나타낸다. 그리고 소스 코드 난독화를 수행하는 방식은 표에서 Obfuscation Method에 해당하며, 이름 맹글링, 더미 데이터 삽입, 유사 데이터 삽입, 데이터 인코딩 방식이 있다. 방식별 소스 코드 난독화를 수행하는 방식에 대한 자세한 설명은 다음과 같다.

Table 3. Obfuscation techniques with components and methods.

No.	Obfuscation Component
1	Variable
2	Function
3	Object

No.	Obfuscation Method
1	Name Mangling
2	Dummy Data Insertion
3	Similar Data Insertion
4	Data Encoding

Method 1. 이름 맹글링(Name Mangling): 코드 구성 요소의 이름을 의미 없는 문자열로 변환하는 방식이다. 이 방식은 분석자의 직관적인 이해를 떨어뜨려 코드의 의도를 파악하기 어렵게 한다.

Method 2. 더미 데이터 삽입(Dummy Data Insertion): 코드 내에 사용되지 않는 불필요한 데이터를 추가하여 인위적으로 복잡성을 증가시키는 방식이다. 이 방식은 분석자가 코드에서 실제 실행과 관련한 요소와 무관한 요소를 구분하기 어렵게 한다.

Method 3. 유사 데이터 삽입(Similar Data Insertion): 실제 데이터와 비슷한 요소를 추가하여 원본 데이터와 혼동되게 하고 코드 분석을 방해하는 방식이다. 더미 데이터와 마찬가지로 코드의 핵심 논리와 데이터를 추적하는 과정을 복잡하게 만든다.

Method 4. 데이터 인코딩(Data Encoding): 중요하거나 민감한 데이터를 인코딩 알고리즘을 통해 변환하여 난독화하는 방식이다. 이 방식은 데이터를 정적 분석으로부

터 숨기고, 실행 시점에서만 복구되도록 설계되어 보안을 강화한다. 단, 본 논문의 실험에서는 Base64를 사용한 인코딩 방식만 사용하였다.

IV. Experiments

본 장에서는 난독화된 코드를 생성하는 실험을 수행하면서 구성한 실험 환경 및 입력한 원본 소스 코드, 실험의 상세 및 결과를 소개한다.

1. Experiments Setup

1.1 Environment

코드 생성을 위해 본 연구에서는 대표적인 생성형 AI 모델 중 하나인 OpenAI의 GPT-4o를 사용하였다. 구현을 위해 OpenAI Python API 라이브러리의 1.52.2버전을 사용하였으며, 최소 요구 사항을 충족하기 위해 Python 3.10 환경에서 실험을 진행하였다.

1.2 Dataset

프레임워크의 입력 데이터를 위해 다섯 가지 대표적인 컴퓨터 알고리즘을 선정하였다. 그래프에서 최단 경로를 계산하는 다익스트라 알고리즘과 효율적인 정렬을 위한 병합 정렬은 각각 탐욕 알고리즘과 분할 정복 기법의 대표적인 사례다. 또한, 제한된 용량의 배낭에 최대 가치를 담는 0-1 배낭 문제 알고리즘과 데이터 비교에서 활용되는 Longest Common Subsequence(LCS) 알고리즘은 동적 계획법의 주요 예시로 포함되었다. 마지막으로는 문자열 검색 분야에서는 활용되는 문자열 검색 알고리즘의 대표적 사례인 Knuth-Morris-Pratt(KMP) 알고리즘을 선택하였다. 선정된 다섯 가지 알고리즘은 Python으로 구현되었으며, 소스 코드 난독화 기법을 실험하기 위해 문자열, 연산자, 변수, 클래스, 함수 등 다양한 코드를 포함하도록 설계되었다.

2. Experiments

2.1 Framework Evaluation

실험을 하기에 앞서 본 연구에서는 크게 두 가지의 연구 질문을 제시하였다. 첫 번째 연구 질문은 “생성형 AI를 사용하여 난독화된 소스 코드를 효과적으로 생성할 수 있는가?”이다. 이는 본 연구에서 제시한 프레임워크가 생성형 AI를 소스 코드 난독화 도구로 활용함에 따라 필히 수반되는 질문으로, 도구 활용자의 의도에 따라 적절한 코드를 생성할 수 있는지에 대한 질문이다. 이를 위해 실험에서는 생성한 코드의 실행 여부, 실행 결과 측면에서 원본 코드와의 동등성, 적용하고자 하는 소스 코드 난독화 기법이 의도대로 반영되었는지 등을 확인하였다. 두 번째 연구 질문은 “난독화된 코드를 생성하기 위해 어느 정도의 비용이 소모되는가?”이다. 앞서 언급한 것처럼 본 연구에서 제시하는 프레임워크는 비용 절감 및 편의성 증대를 달성하기 위해 비용 효율적이어야 할 필요가 있다. 이를 위해 난독화된 코드 생성에 따라 소모된 비용과 도구의 실행 시간을 측정함으로써 질문에 대한 답을 구하였다.

본 연구의 실험은 5가지 원본 소스 코드 입력에 대해 각각 12개의 소스 코드 난독화 기법을 적용하여 총 60개의 난독화된 코드를 생성하는 방식으로 진행하였다. 앞서 설명한 프레임워크와 같이 생성된 코드를 실행하여 얻을 수 있는 결과를 확인하고 이를 반영한 최대 5회의 재요청을 통해 최종적으로 난독화 코드를 생성하도록 하였으며, 코드 실행 결과를 반영하지 않고 단일 요청을 수행하는 경우, 코드 실행 결과 및 소스 코드 난독화 기법 예제를 활용하지 않은 경우, 코드 실행 결과를 반영하되 재요청 횟수를 늘린 경우를 베이스라인으로 설정하여 실험 결과를 비교 및 분석하였다.

Table 4. Evaluation matrices for the experiments.

Evaluation Matrices	Description
Execution Success Rate	The ratio of executable code among the generated code.
Equality	The ratio of code that produces the same output as the original code before obfuscation.
Effectiveness	The ratio of code where the intended obfuscation technique is correctly applied, verified through heuristics
Effort Ratio (ER)	The monetary cost calculated based on token usage from API requests and the program execution time.
Cost	A ratio comparing the complexity of the obfuscated code to the original code, based on Halstead’s Complexity Measures.

2.2 Evaluation Metrics

본 연구에서 사용된 평가 지표는 Table 4에 나타난 다섯 가지이다. 먼저, 실행 성공률(execution success rate)은 프레임워크가 생성한 코드의 실행 여부를 확인하는 필수적인 평가 지표이다. 이는 아래의 수식과 같이 표현할 수 있으며, 이때 N_{total}은 생성된 코드의 개수를 의미하며, $N_{success}$는 그 중 성공적으로 실행된 코드의 개수를 의미한다.

$$Execution\,Success\,Rate = \frac{N_{success}}{N_{total}} \times 100$$

또한, 동등성(equality)은 난독화 이전의 코드와 동일한 출력을 생성하는지를 확인함으로써 소스 코드 난독화 적용 여부와 상관없이 프로그램의 기능이 유지되는지를 평가한다. 이는 아래와 같은 수식에 따라 계산될 수 있으며, 이때 N_{equal}은 원본 코드와 동일한 결과를 출력한 코드의 개수를 의미한다.

$$Equality = \frac{N_{equal}}{N_{total}} \times 100$$

반면, 유효성(effectiveness)은 제시한 소스 코드 난독화 기법이 원본 코드에 의도한 대로 적용되었는지 휴리스틱을 통해 확인함에 따라 생성된 코드가 유효한지 판별하기 위한 지표이다. 이는 아래와 같은 수식에 따라 표현할 수 있으며, 이때 $N_{applied}$는 사용자의 의도에 따라 난독화 기법이 적용된 것으로 판별된 코드의 수를 의미한다.

$$Effectiveness = \frac{N_{applied}}{N_{total}} \times 100$$

나아가, 소스 코드 난독화의 정도를 간접적이지만 정량적으로 평가하기 위해 Hu 등[25]이 제시한 Effort Ratio(ER)을 사용하였다. ER은 Halstead's Complexity Measures[26]를 바탕으로 생성형 AI를 통해 생성한 코드의 최적화 정도를 평가하기 위해 입력으로 사용된 코드와 최적화된 코드의 Halstead's Complexity Measures를 각각 계산하여 그 비율을 나타낸 것이다. 비록 본 연구에서는 최적화가 아닌 소스 코드 난독화를 수행하였지만, 반대로 소스 코드 난독화의 정도를 정량적으로 평가하기 위한 지표로 활용될 수 있다. 이는 아래의 수식을 통해 계산할 수 있으며, $Effort_{origijnal}$과 $Effort_{generated}$는 각각 원본 소스 코드와 생성된 소스 코드의 Halstead's Complexity Measures를 나타낸다.

$$ER = \frac{Effort_{generated}}{Effort_{original}}$$

마지막으로 금전적 및 시간적 비용을 측정하여 도구의 비용 효율성을 측정하였으며, 이러한 다섯 가지 지표를 바탕으로 실험의 결과를 종합적으로 평가하였다.

2.3 Experiments Results

Fig. 5는 난독화된 코드 생성 실험에 따른 결과를 나타낸 막대그래프이다. 결과에 따르면, 요청을 1회 수행한 경우, 실행 결과를 반영하여 최대 5회 및 10회 재요청한 경우는 유사한 결과를 보였으며, 그중 최대 5회의 재요청을 수행했을 때 가장 높은 지표를 보였다. 이는 반복적인 API 재요청에 대해 요청 횟수가 증가할수록 더 나은 결과를 보장하는 것이 아님을 의미한다. 반면, 예제를 제공하지 않은 경우는 가장 낮은 성능 지표를 보였으며, 유효성에서 특히 낮은 지표를 나타냈다. 이는 생성형 AI에 대한 API 요청 상에서 의도하고자 하는 바를 적절히 수행 하도록 하기 위해, 예시를 함께 제공하는 것에 대한 중요성을 보여준다.

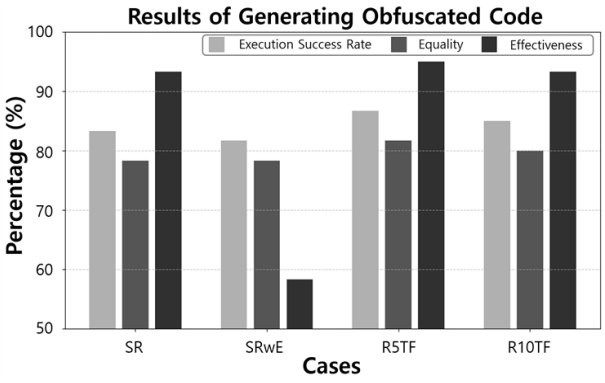

Fig. 5. Results of generating obfuscated code (SR: Single Request, SRwE: Single Request without Example, R5TF: Retry 5 Times based on Feedback, R10TF: Retry 10 Times based on Feedback).

또한, Table 5는 앞서 계산한 성공률, 동등성, 유효성을 종합한 지표를 나타낸 것이다. 즉, 생성한 코드 중 실행이 가능하면서 원본 코드와 결과가 동일할 뿐만 아니라 의도에 맞게 소스 코드 난독화 기법이 적용된 비율을 나타낸다. 또한, Table 5는 앞서 계산한 성공률, 동등성, 유효성을 종합한 지표를 나타낸 것이다. 즉, 생성한 코드 중 실행이 가능하면서 원본 코드와 결과가 동일할 뿐만 아니라 의도에 맞게 소스 코드 난독화 기법이 적용된 비율을 나타낸다.

Table 5. Success rates of generating obfuscated code by using GPT-4o.

Cases		Success Rate
SR		76.67%
SRwE		53.33%
R5TF	GPT-3.5-turbo	3.33%
	GPT-4-turbo	23.33%
	GPT-4o	80.00%
R10TF		78.33%

이 중 가장 높은 지표를 보인 것은 앞선 지표와 동일하게 실행 결과에 따라 최대 5회의 재요청(R5TF)을 수행한 경우였으며, 예제를 제공하지 않는 경우 제일 낮은 지표를 보였다. 나아가 생성된 코드를 확인한 결과, 데이터 인코딩을 적용하도록 요청한 상당수의 코드는 실행되지 않은 것으로 확인되었으며, 특히 그중 일부는 인코딩되기 이전의 평문을 소스 코드상에 그대로 노출하는 경우 역시 존재하였다.

생성형 AI 모델에 따른 성능 비교를 위해 성능이 가장 좋은 R5TF에 대해 GPT-3.5-trubo 및 GPT-4-turbo 모델을 추가적으로 분석하였다. 난독화 적용에는 모델에 따른 성능 차이가 가장 큰 것을 확인하였다. GPT-3.5-turbo(성공률: 3.33%)와 GPT-4-turbo(성공률: 23.33%)의 경우 난독화 기법을 제대로 적용하지 못하였으며 생성된 대부분의 코드가 실행되지 않았다.

Table 6. Average and standard deviation of obfuscated code's ER (μ: mean, σ: standard deviation, w. de: with data encoding, w/o. de: without data encoding).

Cases	μ (w. de)	σ (w. de)	μ (w/o. de)	σ (w/o. de)
SR	1.2161	0.7111	1.3244	0.5664
SRwE	1.1969	0.5522	1.2144	0.4962
R5TF	1.2656	0.9277	1.3852	0.8772
R10TF	1.2841	0.7899	1.3880	0.6931

나아가, Table 6은 프로그램 실행 결과 생성된 코드별 ER의 평균을 계산한 것이다. 단, 데이터 인코딩 기법의 경우 소스 코드에 드러나는 연산자의 수가 크게 줄어들기 때문에 이를 제외한 지표를 함께 계산하여 확인하였다. 확인 결과, 난독화된 코드의 연산 횟수는 원본 코드의 약 1.2배에서 1.3배 수준인 것으로 확인이 되며, 예제를 제공하지 않은 경우를 제외하면 서로 유사한 수준으로 연산 횟수가 증가한 것을 확인할 수 있다. 또한, 재요청을 수행하는 경우가 단일 요청을 수행하는 경우보다 연상 횟수가 높은 것으로 나타나는데, 이는 실행 결과를 확인하고 응답을 수정하는 과정에서 추가적인 코드를 삽입함에 따른 영향에 의한 것으로 보여진다. 반면, 예제를 제공하지 않는 경우엔 모델이 비교적 상세하지 않은 코드를 제공함에 따라 연산 횟수의 증가량이 다른 접근 방식에 비해 떨어지는 것을 확인할 수 있다.

Fig. 6은 원본 코드 중 하나인 다익스트라 알고리즘 구현과 프레임워크에 따라 난독화된 코드의 예시를 나타낸 것이다. 먼저, 이름 맹글링의 경우, 원본 코드의 함수명인 add_edges를 a2b3c4로 변경하였으며, 더미 데이터 삽입에선 원본 코드에 없었던 딕셔너리 및 바이트 타입의 변수를 새로 정의하였다. 또한, 유사 데이터 삽입의 경우, 기존의 dijkstra 함수 대신 dijikstra라는 함수를 추가하였으며, 이는 각 노드의 초기값을 0으로 설정하는 등 기존 코드와 다르게 동작하도록 구현된 것을 확인할 수 있다. 이는 단순히 이름이 유사한 함수를 추가하는 것뿐만 아니라, 생성형 AI 바탕의 프레임워크를 사용할 경우, 비교적 정교하게 소스 코드 난독화를 적용할 수 있는 것을 보여준다.

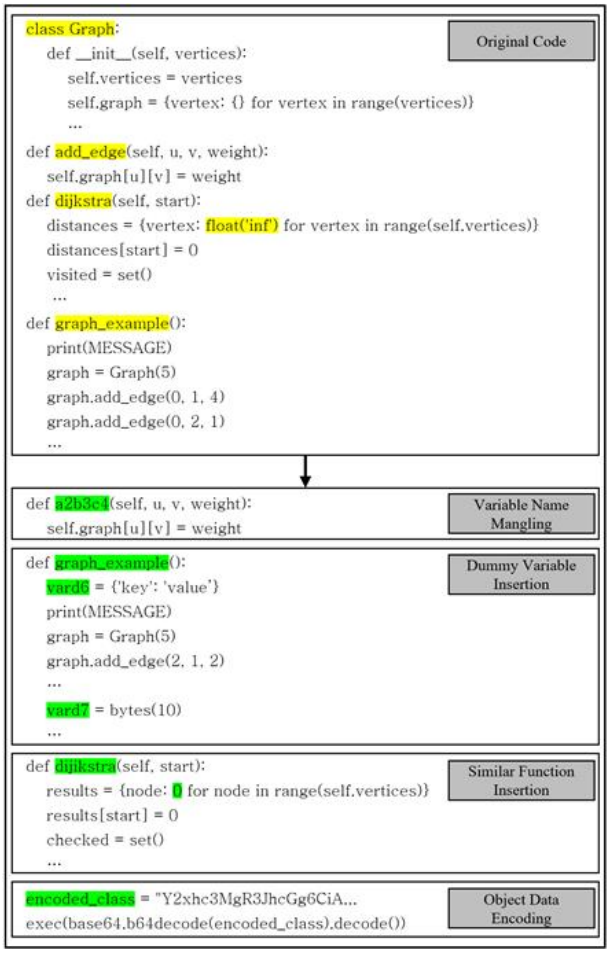

Fig. 6. Example of Generated Codes with the Obfuscated Techniques.

또한, 데이터 인코딩은 기존의 Graph 클래스 자체를 Base64로 인코딩하였고, 실제 실행할 때는 디코딩한 후 실행해야 하도록 함으로써 코드 분석을 어렵게 할 수 있을 것으로 보여지며, 적절한 소스 코드 난독화가 수행되었음을 확인할 수 있다.

Table 7은 난독화된 코드 생성 과정 중 API 요청에 따라 발생한 금전적 비용과 프로그램 실행 시간을 나타낸 것이다. 총 60개의 알고리즘 구현에 대한 소스 코드 난독화를 수행하는 과정에서 지출된 비용은 약 1달러 수준이며, 수행 시간은 10-20분 정도이다. 이는 앞서 확인한 난독화 수준을 생각했을 때 본 연구에서 제시한 프레임워크가 충분히 비용 효율적인 수단으로 활용될 여지가 있다는 것을 보인다. 또한, 단일 요청과 최대 5회 및 10회 재요청하는 경우, 금전적 비용과 실행 시간은 비례하여 증가하는 것을 확인할 수 있으며, Fig. 5와 Table 5에서 확인한 결과를 고려해 보았을 때, 최대 재요청 횟수를 늘리는 것보다 더 적은 API 요청을 수행했을 때, 보다 효율적으로 난독화된 코드를 생성하는 것을 확인하였다.

Table 7. Cost and execution time of generating obfuscated code.

Cases	Cost ($)	Execution Time (sec)
SR	0.44	678
SRwE	0.43	603
R5TF	0.91	1,214
R10TF	1.90	2,342

본 연구에서는 제안한 프레임워크의 난독화 성능을 평가하기 위해 기존 오픈소스 도구인 Pyminifier와 Opy로 난독화된 소스 코드를 비교 분석하였다. 난독화 기법별 난독화 수준을 비교하기 위해 코드 변환의 유사성을 측정하는 지표인 BLEU score[29]를 사용하였으며, BLEU 값이 낮을수록 난독화가 더 잘 이루어진 것으로 판단하였다.

기존 오픈소스 도구는 미리 정의된 난독화 기법을 모두 적용하기 때문에, 공정한 비교를 위해 난독화가 적용된 부분을 식별한 후, 각 난독화 기법에 해당하는 부분만을 남겨두는 방식으로 실험을 진행하였다. 또한, 각 도구가 공통적으로 지원하는 난독화 기법에 대해서만 실험을 수행하였으며, 제안한 프레임워크에서만 제공하는 난독화 기법은 비교 대상에서 제외하였다.

Table 8. Average BLEU scores of each model categorized by different techniques.

Models	Variable Name Mangling	Function Name Mangling	Object Name Mangling	String Encoding
Pyminifier [27]	0.3464	0.4588	0.3564	-
Opy [28]	0.4417	0.8216	0.4643	0.9115
Ours	0.3803	0.5923	0.3616	0.9361

Table 8에 제시된 BLEU 점수를 바탕으로 통계 테스트를 수행한 결과, Variable Name Mangling ($p = 0.5391$)과 Object Name Mangling ($p = 0.3844$) 기법에서는 ANOVA 테스트를 통해 통계적으로 유의미한 차이가 나타나지 않았다. 반면, Function Name Mangling 기법에서는 통계적으로 유의미한 차이가 확인되었으며($p < 0.05$), 사후 검정으로 Bonferroni correction 테스트를 수행한 결과, Pyminifier와 Ours는 유의미한 차이가 나타나지 않았으나($p = 0.0674$), Opy는 다른 두 기법과 통계적으로 유의미한 차이를 보였다 ($p < 0.05$). String Encoding 기법에서는 두 기법(Opy와 Ours)에 대해 t-test를 수행한 결과, 통계적으로 유의미한 차이가 확인되었다($p < 0.05$).

결론적으로, 제안하는 프레임워크는 기존 오픈소스 난독화 도구와 비교하여 성능 면에서 큰 차이를 보이지 않는 것으로 확인되었다.

V. Conclusions

본 논문에서는 GPT-4o를 이용하여 소스 코드 난독화를 자동화하는 프레임워크를 제안했다. 기존 난독화 작업의 유연성과 효율성 부족 문제를 해결하기 위해 12개의 난독화 기법을 기반으로 GPT-4o에 입력으로 사용할 프롬프트를 생성하고 이를 통해 난독화된 소스 코드를 자동으로 생성하는 프레임워크를 구현했다. 제안된 프레임워크는 생성된 난독화 소스 코드를 자동으로 실행하고 검증하는 과정을 포함하여 난독화된 소스 코드가 원본과 동일한 의도를 유지하는지 검증하였다. 제안된 프레임워크는 난독화된 소스 코드의 80%가 성공적으로 실행되었으며 난독화 기법도 효과적으로 적용되었음을 확인했다.

향후 연구에서는 제안된 프레임워크를 확장하여 다양한 프로그래밍 언어와 난독화 기법에 적용 가능성을 확인하여 범용성을 확보하고, 여러 난독화 기법을 조합하여 복잡성을 높이려고 한다.

ACKNOWLEDGEMENT

This work was supported by a Korea University Grant. Also, this work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT(Ministry of Science and ICT)) (No. NRF-00252157).

REFERENCES

[1] N. Zhang, W. Dong, Z. Xia, Z. Feng, and Z. Li, “Analysis on Technique for Code Obfuscation,” In Proceedings of the 2023 2nd International Conference on Networks, Communications and Information Technology, pp. 241-245, 2023.

[2] S. A. Ebad, A. A. Darem, and J. H. Abawajy, “Measuring software obfuscation quality-a systematic literature review,” IEEE Access, Vol. 9, pp.99024-99038, 2021, doi: 10.1109/ACCESS.2021.3094517.

[3] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, ..., and J. R. Wen, “A survey of large language models,” arXiv preprint arXiv:2303.18223, 2023.

[4] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, “A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly,” High-Confidence Computing, Vol. 4, Issue

2, 2024, 100211.

[5] P. Hacker, A. Engel, and M. Mauer, “Regulating ChatGPT and other large generative AI models,” In Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency, pp.1112-1123, June. 2023.

[6] R. Gozalo-Brizuela, and E C. Garrido-Merchan., "ChatGPT is not all you need. A State of the Art Review of large Generative AI models," arXiv preprint arXiv:2301.04655, 2023.

[7] A. Casheekar, A. Lahiri, K. Rath, K. S. Prabhakar, and K. Srinivasan, “A contemporary review on chatbots, AI-powered virtual conversational agents, ChatGPT: Applications, open challenges and future research directions,” Computer Science Review, Vol. 52, 2024, 100632.

[8] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” arXiv preprint arXiv:2402. 07927, 2024.

[9] B. Chen, Z. Zhang, N. Langrené, and S. Zhu,, “Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review,” arXiv preprint arXiv:2310.14735, 2023.

[10] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, ..., and Y. Zhang, “Sparks of artificial general intelligence: Early experiments with gpt-4,” arXiv preprint arXiv:2303.12712, 2023.

[11] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” ACM Computing Surveys, Vol. 55, No.9, pp.1-35, 2023.

[12] G. Vishwas, N. S. Nithin, P. Varshith, and M. Belwal, “Unveiling the World of Code Obfuscation: A Comprehensive Survey,” 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp.1-8, Bangalore, India, 2023, doi: 10.1109/CSITSS60515.2023. 10334127.

[13] A. Pawlowski, M. Contag, and T. Holz, "Probfuscation: an obfuscation approach using probabilistic control flows," Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13. Springer International Publishing, 2016.

[14] N. Zhang, Z. Feng, and D. Xu, "An In-Depth Analysis of the Code-Reuse Gadgets Introduced by Software Obfuscation," International Conference on Applied Cryptography and Network Security. Cham: Springer Nature Switzerland, 2024.

[15] J. Jiang, F. Wang, J. Shen, S. Kim and S. Kim, “A Survey on Large Language Models for Code Generation,” DOI: 10.48550/ arXiv.2406.00515, Jun. 2024.

[16] OpenAI, "Evaluating Codex with HumanEval," OpenAI Research Blog, 2021.

[17] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation,” Oct. 2023. DOI: 10.48550/arXiv.2305.01210.

[18] T. Y. Zhuo, et al., “BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions,” Oct. 2024. DOI: 10.48550/arXiv.2406.15877.

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Proc. of EMNLP, 2020. DOI: 10.48550/arXiv.2002.08155.

[20] Y. Wang, W. Wang, S. Joty, and S. C.H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in Proc. of ACL, 2021. DOI: 10.48550/arXiv.2109.00859.

[21] W. Zaremba, G. Brockman, and OpenAI, “OpenAI Codex,” https://openai.com/index/openai-codex/

[22] Y. Wei, et al., “Magicoder: Empowering Code Generation with OSS-Instruct,” Jun. 2024. DOI: 10.48550/arXiv.2312.02120.

[23] N. Alshahwan, et al., “Automated Unit Test Improvement using Large Language Models at Meta,” Feb. 2024. 10.48550/arXiv. 2402.09171.

[24] D. Huang, et al., “EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization,” Oct 2024. DOI: 10.48550/arXiv. 2405.15189.

[25] P. Hu, R. Liang, and K. Chen, “DeGPT: Optimizing Decompiler Output with LLM,” Network and Distributed System Security (NDSS) Symposium 2024, San Deigo, CA, USA, Feb. 2024. DOI: 10.14722/ndss.2024.24401

[26] M. H. Maurice, “Elements of Software Science (Operating and programming systems series),” Elsevier Science Inc., 1977.

[27] “pyminifier.” [Online]. Available: https://liftoff.github.io/pymin ifier/.

[28] “Opy.” [Online]. Available: https://github.com/QQuick/Opy.

[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A Method for Automatic Evaluation of Machine Translation,” in Proc. of the 40th Annual Meeting of the Association for Computational Linguistics, pages 311−318, 2002.

Authors

Jihun Han is a undergraduate student of Department of AI Cyber Security at Korea University, Sejong. He is interested in natural language processing and AI-based cyber security.

Seung-A Park received the B.S. degree in Department of AI Cyber Security from Korea University, Sejong, in 2023. She is currently pursuing M.S. degree in the Department of Cyber Security, Korea University. She is interested in design and analysis of information security protocols, and AI security.

Joonseo Ha received the B.S. degree in Department of AI Cyber Security from Korea University, Sejong, in 2023. He is currently pursuing M.S. degree in the Department of Cyber Security, Korea University. He is interested in network security, machine learning, and data science.

Chang-min Lee is a undergraduate student of Department of AI Cyber Security at Korea University, Sejong. He is interested in reverse engineering, and system hacking.

Kyung-mi Jung received the B.S. and M.S. degrees in Statistics from Kyungpook University, Korea, in 1995 and 1997, respectively. She is currently a researcher in Edge Cloud Data Security Research Center, Korea University, Sejong. Her research focuses on machine learning and differential privacy.

Mee Lan Han received the B.S. degree in Computer Science from Dongduck Women's University in 2002, and the M.S., Ph.D. degrees in Cyber Security from Korea University in 2015, 2020, respectively. She joined the faculty of the Department of AI Cyber Security at Korea University, Sejong, Korea, in 2022. Prior to joining the university, she worked as a senior researcher at Chinese-speaking countries Development Part, Nexon Korea and as a research professor at Institute of Cyber Security & Privacy, Korea University. She is interested in Anomaly detection and identification, criminal behavior analysis, and embedded security.

Jun-Seob Kim received the B.S. degree in Computer Engineering from Chungbuk National University in 1997, the M.S. degree in Information Security from Sejong University in 2022, respectively. He joined the faculty of the Department of AI Cyber Security at Korea University, Sejong, Korea, in 2022. He is currently a Professor in the Department of AI Cyber Security at Korea University. He is interested in malware analysis, and endpoint security.

Geumhwan Cho received his B.S. degree in Information and Communication Engineering from Cheongju University in 2011, his M.S. degree in Computer Engineering from Kyunghee University in 2013, and his Ph.D. degree in Computer Science from Sungkyunkwan University in 2020. He joined the Department of AI Cyber Security as a faculty member in 2024 and currently serves as an Assistant Professor in the department. Prior to joining the university, he worked as a senior researcher at National Security Research Institute. His research interests focus on usable security and digital forensics with AI.