The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 24, No. 4, pp.49-55, Aug. 31, 2024. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2024.24.4.49
JIIBC 2024-4-8

# LEA 코드를 위한 코드 스멜 관점에서 메트릭 접근

# Metrics Approach in aspect of Code Smell for LEA Code

홍진근[*]

Jin-Keun Hong[*]

**요 약** 코드 스멜은 Kent Beck에 의해 사용된 개념으로, 잠재적인 품질 문제를 나타내며 리팩토링의 필요성을 제시한다. 본 논문은 LEA 코드베이스에서 코드 스멜을 평가하며, 분류와 관련된 메트릭에 중점을 둔다. 연구에서는 LEA_core.c와 LEA.cpp를 분석하여 코드 품질과 복잡성의 차이를 강조한다. 또한 연구에서는 LOC, NOM, NOA, CYCLO, MAXNESTING, FANOUT와 같은 메트릭을 사용하여 크기, 복잡성, 결합도, 캡슐화, 상속, 응집도를 평가한다. 연구 결과에서는 LEA_core.c가 LEA.cpp에 비해 더 복잡하고 유지보수가 어려운 것으로 나타났다. 우리는 향후 연구에서 실시간 코드 스멜 탐지 및 리팩토링 제안을 위한 자동화 도구를 개발할 것이다.

**Abstract** Code smells, used by Kent Beck, indicate potential quality issues and suggest the need for refactoring. This paper evaluates code smells in the LEA codebase, focusing on categorization and associated metrics. The research analyze LEA_core.c and LEA.cpp, highlighting differences in code quality and complexity. And metrics such as LOC, NOM, NOA, CYCLO, MAXNESTING, and FANOUT are used to assess size, complexity, coupling, encapsulation, inheritance, and cohesion. In the result of research, LEA_core.c is found to be more complex and challenging to maintain compared to LEA.cpp. In future work, we will develop automated tools for real-time code smell detection and refactoring suggestions

**Key Words :** Code, Smell, Metrics, Categories, Crypto

## Ⅰ. 서 론

최근 머신러닝에 대한 연구가 활발한데, 머신러닝 알고리즘 또한 코드 베이스로 이루어진다 [1]. 코드 스멜(Code smell)은 Kent Beck에 의해 사용된 용어로, 코드 품질에 잠재적인 문제가 있음을 나타내며, 리팩토링이 필요함을 뜻하는 특정 코드 구조이다 [2]. 코드 스멜은 소스 코드 내 문제나 잠재적 결함을 나타내는 지표이다. 이는 코드 설계에 결함이 있거나 개선이 필요함을 암시하며, 코드의 이해와 유지보수를 어렵게 만들 수 있다. 코드 스멜은 크기, 복잡성, 결합도, 캡슐화, 상속, 응집도와 같은 다양한 차원으로 분류될 수 있다.

본 논문은 LEA 코드베이스의 클래스에서 코드 스멜을 평가하며, 그 분류와 관련된 메트릭에 중점을 둔다. 2장에서는 관련 연구를 검토하고, 3장에서는 코드 스멜을 평가하는 데 사용된 분류와 메트릭을 살펴본다. 4장에서는 결론을 제시한다.

[*]정회원, 백석대학교 첨단IT학부
접수일자 2024년 6월 19일, 수정완료 2024년 7월 19일
게재확정일자 2024년 8월 9일

Received: 19 June, 2024 / Revised: 19 July, 2024 /
Accepted: 9 August, 2024
[*]Corresponding Author: jkhong@bu.ac.kr
Division of Advanced IT, Baekseok University, Korea

# II. 관련 연구

다양한 연구들이 코드 스멜과 리팩토링을 탐구하며, 코드 스멜의 정의와 그 의미에 대한 중요한 통찰을 제공하고 있다. 특히, Sharma와 Kessentini는 대규모 데이터셋을 대상으로 코드 스멜과 품질 메트릭에 대한 광범위한 리뷰를 수행하고 있다 [3]. 또한, Agnihotri와 Chug는 소프트웨어 메트릭, 코드 스멜, 리팩토링 기법에 대한 종합적인 검토를 제시하고 있다 [4].

T. Sharma의 연구와 같이, 코드 스멜을 탐지하기 위한 연구에는 서포트 벡터 머신과 이진 로지스틱 회귀 모델을 사용하는 접근 방식이 포함된다 [5]. 딥러닝 분야에서는 합성곱 신경망(CNN)과 순환 신경망(RNN)이 코드 스멜 탐지에 적용되었다 [6]. K. Das 등은 딥러닝을 활용하여 "브레인 클래스"와 "브레인 메소드"라는 코드 스멜을 식별하고 있다 [7]. 소스 코드에서 코드 스멜은 소프트웨어의 아키텍처, 설계 및 코드에서 증상으로 나타난다. 코드 스멜 탐지는 소프트웨어 리팩토링 프로세스의 중요한 요소이다.

코드의 스멜 탐지는 일반적으로 특정 규칙이나 표준에 의존하고 리팩토링을 요구한다 [3][8].

H. Liu 등은 다중 레이블 코드 스멜을 탐지하기 위해 프롬프트 학습 기반의 새로운 접근 방식인 \textit{PromptSmell}을 제안하고 있다 [2].

Lee 등은 딥러닝에서 버퍼 캐시 성능 향상에 대한 연구를 하였다 [9]. 버퍼 캐시 문제는 코드 기반의 효율적인 메모리 관리와 관련되는데, 연구에서는 딥러닝 환경에서 캐싱의 어려움을 해결하기 위해 접근한다.

Kim 등은 이상치와 결측치 보정에 대해 연구하였다 [10]. IoT 센서 기반의 환경 데이터를 수집에서 이상치와 결측치에 대한 전처리를 데이터 예측 품질에 매우 심각한 영향을 미칠 수 있다. 미세한 파라미터의 조정은 결과에 심각한 영향을 줄 수 있으므로 빅데이터 예측 분석 과정에서 일어나는 파인 튜닝 과정에서 적절한 코드 함수의 사용과 효율적인 메모리 관리는 민감한 문제이다. Choi 등은 셀 브로드캐스팅 기술 활용에 위치 기반의 보안인증에 대한 연구를 수행하였다 [11]. 이 연구에서의 위치 기반 정보나 보안 인증 정보 또한 버퍼 메모리 관리 측면에 이들 데이터는 민감한 문제이다.

Kang 등은 이더리움 온 체인 피싱 탐지방안에 대해 연구를 수행하였다 [12]. 이 연구는 피싱 활동 탐지를 어텐션 기반 ID CNN 모델을 제안하는데, 피싱 탐지에 가장 중요한 부분은 데이터 모델이다. 이들 중요 데이터에 대한 보안 처리에 있어, 중요 데이터 관리에 사용되는 경량 암호 알고리즘 LEA에 대한 코드 품질관리는 민감한 문제이다.

본 논문에서는 LEA_core.c 암호화 코드의 특정 문맥 내에서 코드 스멜을 탐지하기 위한 관련 메트릭을 분류하고 적용할 것이다.

# III. 코드 스멜 평가를 위한 메트릭

## 1. LEA_core.c와 LEA.cpp 코드 구조

LEA_core.c와 LEA.cpp 코드들은 LEA 암호화 알고리즘을 구현하기 위한 다양한 함수를 포함하고 있다. 이 코드들은 알고리즘에 필수적인 상수 값을 정의하고 비트 회전 함수(ROL, ROR)를 지원한다. 또한, 암호화 및 복호화 과정에 필요한 라운드 키를 생성하기 위해 키 스케줄링 함수(KeySchedule_128)를 사용한다.

크기, 복잡성, 결합도, 캡슐화, 상속, 응집도와 같은 범주(category)는 코드 스멜을 평가하는 데 대표적이다. 각 범주를 평가하는 데 사용되는 메트릭은 다음에서 제시하였다.

## 2. LEA_core.c와 LEA.cpp 코드 스멜 비교

코드 스멜은 다음 범주와 해당 식별에 사용되는 메트릭을 통해 분류할 수 있다.

### 가. 크기(Size)

크기 관련 코드 스멜은 지나치게 길어 이해하기 어렵고 유지보수가 어려운 메서드나 클래스를 나타낸다.

코드 라인 수(Lines of Code): LEA_core.c 및 LEA.cpp 코드의 총 라인 수를 측정한다.

메서드 수(Number of Methods): LEA_core.c 및 LEA.cpp의 클래스나 모듈 내에 정의된 메서드 수를 측정한다.

속성 수(Number of Attributes): LEA_core.c 및 LEA.cpp의 클래스 내에 정의된 속성(필드) 수를 측정한다.

### 나. 복잡성(Complexity)

복잡한 코드는 이해하기 어렵고 오류가 발생하기 쉽다. 복잡성은 중첩된 조건문, 과도한 분기, 복잡한 표현식 등에서 나타난다.

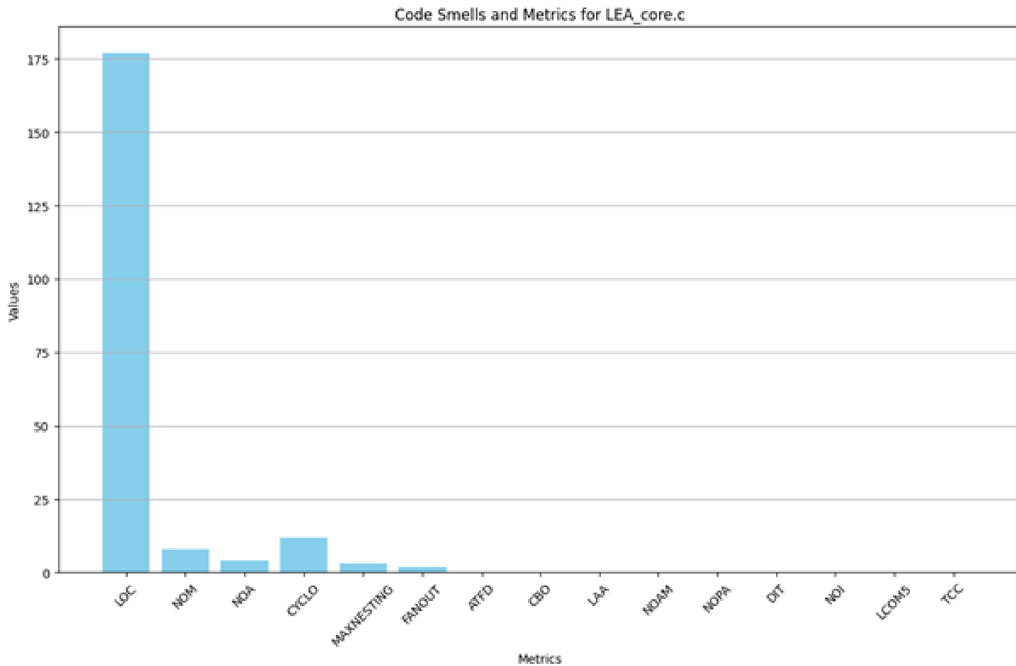


**그림 1. LEA_core.c 코드의 메트릭**
**Fig. 1. Metrics of LEA_core.c code**

사이클로매틱 복잡도(Cyclomatic Complexity)은 LEA_core.c 및 LEA.cpp의 코드에서 선형적으로 독립된 경로의 수를 측정한다(예: 조건문 포함).

클래스당 가중 메서드(Weighted Methods per Class)는 LEA_core.c 및 LEA.cpp 내의 모든 메서드의 복잡도를 합산하여 나타낸다.

최대 중첩 수준(Maximum Nesting Level)는 LEA_core.c 및 LEA.cpp의 메서드 내에서 가장 깊은 중첩 수준을 측정한다.

## 다. 결합도(Coupling)

높은 결합도는 다른 모듈이나 클래스에 대한 강한 의존성을 가지며, 유연성을 줄이고 변경을 어렵게 만든다.

팬 아웃 복잡도(Fan Out Complexity)은 LEA_core.c 및 LEA.cpp 클래스나 모듈이 의존하는 다른 클래스나 모듈의 수를 측정한다. 높은 값은 시스템의 다른 부분의 변경이 코드에 영향을 미칠 가능성이 높음을 나타낸다.

외부 데이터 접근(Access to Foreign Data)은 LEA_core.c 및 LEA.cpp의 클래스가 외부 클래스의 필드에 접근하는 횟수를 측정한다. 높은 ATFD는 높은 결합도를 나타낸다.

객체 간 결합도(Coupling Between Objects)는 LEA_core.c 및 LEA.cpp의 클래스와 다른 클래스 간의 상호작용 정도를 측정한다. 높은 CBO 값은 높은 결합도를 나타낸다.

## 라. 캡슐화(Encapulation)

속성 접근의 지역성(Locality of Attribute Accesses)은 LEA_core.c 및 LEA.cpp 내에서 속성 접근의 지역성을 측정하여 속성이 클래스 내에서 얼마나 접근되는지를 나타낸다. 현재 속성 사용 분석이 수행되지 않아 0으로 나타낸다.

접근자 메서드 수(Number of Accessor Methods)는 LEA_core.c 및 LEA.cpp에 정의된 접근자 메서드(게터와 세터)의 수를 측정한다. 현재 접근자 메서드 식별이 수행되지 않아 0으로 나타낸다.

공용 속성 수(Number of Public Attributes)는 LEA_core.c 및 LEA.cpp에 정의된 공용 속성의 수를 측정한다. 현재 공용 속성 식별이 수행되지 않아 0으로 나타낸다.

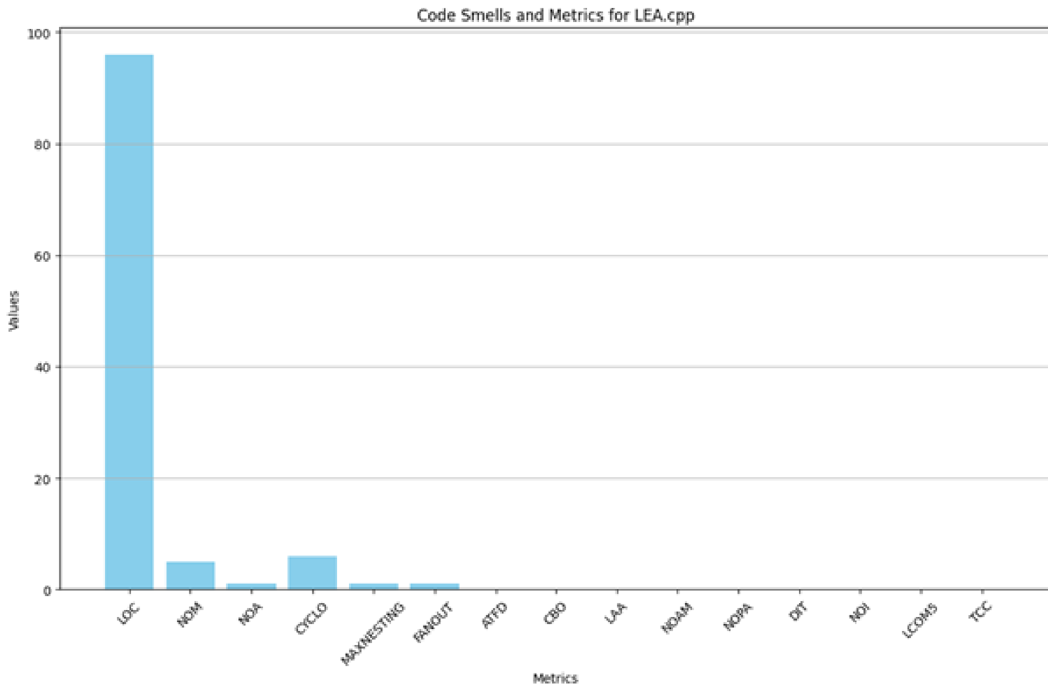


**그림 2. LEA.cpp의 메트릭**
**Fig. 2. Metrics of LEA.cpp**

### 마. 상속성(Inheritance)

상속 트리 깊이(Depth of Inheritance Tree)는 LEA_core.c 및 LEA.cpp의 상속 트리 깊이를 측정한다. 현재 상속 트리 분석이 수행되지 않아 0으로 나타낸다.

구현된 인터페이스 수(Number of Implemented Interfaces)는 LEA_core.c 및 LEA.cpp가 구현한 인터페이스의 수를 측정한다. 현재 인터페이스 분석이 수행되지 않아 0으로 나타낸다.

### 바. 응집성(Cohesion)

메서드 간 응집도 부족(Lack of Cohesion in Methods)은 LEA_core.c 및 LEA.cpp의 메서드 간 응집도 부족 정도를 측정한다. 현재 메서드 응집도 분석이 수행되지 않아 0으로 나타낸다.

타이트 클래스 응집도(Tight Class Cohesion)는 LEA_core.c 및 LEA.cpp의 메서드가 얼마나 밀접하게 관련되어 있는지를 측정한다. 현재 메서드 응집도 분석이 수행되지 않아 0으로 나타낸다.

그림 1과 그림 2에서, 코드 라인 수(LOC)에 서 'LEA_core.c' 코드는 'LEA.cpp' 코드에 비해 상당히 높은 라인 수를 나타내며, 이는 'LEA_core.c'에 더 많은 코드 양이 포함되어 있음을 뜻한다.

메서드 수(NOM)의 경우, 'LEA_core.c'는 많은 메서드를 포함하고 있어, 'LEA.cpp'에 비해 더 높은 함수 또는 절차의 밀도를 나타낸다.

그림 1에서 숫자 카운트는 다음과 같다: LOC(177), NOM(8), NOA(4), CYCLO(12), MAXNESTING(3), FANOUT(2), ATFD(0), CBO(0), LAA(0), NOAM(0), NOPA(0), DIT(0), NOI(0), LCOM5(0), TCC(0).

그림 1에서 속성 수(NOA)를 고려할 때, 'LEA_core.c'는 더 많은 속성을 보유하고 있으며, 이는 더 많은 변수나 필드를 포함하고 있다.

그림 1과 그림 2에서, 사이클로매틱 복잡도(CYCLO)를 검토한 결과, 'LEA_core.c'는 더 높은 사이클로매틱 복잡도를 나타내며, 이는 코드 내 더 복잡한 제어 흐름과 증가된 결정 지점을 의미한다.

최대 중첩 수준(MAXNESTING)에 관해서는, 'LEA_core.c'가 더 큰 최대 중첩 깊이를 보여주며, 이는 코드 블록의 더 빈번하고 깊은 중첩을 나타낸다. 이 분석으로부터, 'LEA_core.c'는 더 많은 코드 양, 더 많은 함수, 더 많은 속성, 증가된 복잡성, 깊은 중첩 수준을 특징으로 하여 'LEA.cpp'에 비해 본질적으로 더 복잡하고 더

**표 1. LEA_core.c와 LEA.cpp의 메트릭**
**Table 1. Metrics of LEA_core.c and LEA.cpp**

| Source Code | LOC | NOM | NOA | CYCLO | MAXNESTING | FANOUT |
|---|---|---|---|---|---|---|
| LEA_core.c | ≈177 | ≈8 | ≈4 | ≈12 | ≈3 | ≈2 |
| LEA.cpp | ≈100 | ≈6 | ≈1 | ≈8 | ≈2 | ≈1 |

광범위한 기능을 포함하고 있음을 알 수 있다.

반면에 'LEA.cpp'는 더 간결한 구조를 가지며, 코드 라인 수, 메서드 수, 속성 수가 적고, 복잡성이 낮으며, 중첩 수준이 얕아 더 단순하고 간소화된 코드베이스를 나타낸다.

표 1의 코드 라인 수( Lines of Code)의 경우, 'LEA_core.c' 파일은 약 177개의 코드 라인으로 구성되어 있으며, 반면 'LEA.cpp' 파일은 약 100개의 코드 라인을 포함하고 있다. 이 메트릭은 'LEA_core.c'가 상당히 더 많은 코드 양을 포함하고 있으며, 이는 더 광범위한 구현을 나타내며 추가적인 기능과 기능을 포함할 가능성이 있음을 의미한다.

메서드 수(NOM) 측면에서, 'LEA_core.c'는 대략 8개의 메서드를 포함하고 있는 반면, 'LEA.cpp'는 약 6개의 메서드를 포함하고 있다. 'LEA_core.c'의 더 많은 함수 수는 더 넓은 범위의 작업을 수행함을 나타내며, 이는 코드 내에서 더 높은 수준의 복잡성에 기여한다.

속성 수(NOA)에서 'LEA_core.c' 파일은 약 4개의 속성을 가지고 있는 반면, 'LEA.cpp' 파일은 약 1개의 속성을 포함하고 있다.

이러한 차이는 'LEA_core.c'가 변수나 필드를 통해 더 많은 상태 정보를 관리하며, 따라서 코드 내 데이터 관리의 복잡성을 증가시킨다는 것을 나타낸다.

사이클로매틱 복잡도 (CYCLO)에서 'LEA_core.c'의 사이클로매틱 복잡도는 약 12인 반면, 'LEA.cpp'는 약 8의 사이클로매틱 복잡도를 보인다. 이 메트릭은 소스 코드의 선형적으로 독립된 경로의 수를 측정하며, 제어 흐름의 복잡성을 반영한다.

'LEA_core.c'의 더 높은 사이클로매틱 복잡도 값은 추가적인 조건 분기와 결정 지점을 포함한 더 복잡한 제어 구조를 뜻한다.

최대 중첩 수준(MAXNESTING)에서 'LEA_core.c'는 약 3의 최대 중첩 수준을 보이는 반면, 'LEA.cpp'는 약 2의 최대 중첩 수준을 보인다. 이 메트릭은 루프나 조건문과 같은 중첩된 제어 구조의 깊이를 나타낸다.

'LEA_core.c'의 더 큰 중첩 수준은 더 깊이 중첩된 논리를 나타내며, 이는 코드의 가독성과 유지보수를 복잡하게 만들 수 있다.

팬아웃 복잡도(FANOUT)에서 'LEA_core.c'는 약 2의 팬아웃 복잡도를 보이는 반면, 'LEA.cpp'는 약 1의 팬아웃 복잡도를 보인다. 팬아웃 복잡도는 특정 모듈이나 함수가 호출하는 다른 모듈이나 함수의 수를 측정한다. 'LEA_core.c'의 더 높은 팬아웃 값은 외부 모듈이나 함수와의 상호작용 정도가 더 크다는 것을 나타내며, 이는 더 높은 결합도와 잠재적으로 다른 시스템 구성 요소에 대한 더 많은 의존성을 의미한다.

LEA_core.c 코드와 LEA.cpp 코드의 특징을 다음 표 2에서 비교하였다.

LEA.core.c 코드는 직접 메모리 접근과 단순 루프 구조를 사용함으로써, 간결하고, 효율화된 반복 구조, 높은 이식성을 제공하는 코드 구조를 가진다.

LEA.cpp 코드는 객체 지향 기능을 통해 클래스와 멤버 함수를 정의하고 있다. 이 구조는 코드 재사용성은 물론이고 유지보수를 용이하게 한다. 메모리의 자동 관리를 위해 vector, iostream 표준 라이브러리를 사용하여 데이터 관리와 입출력 처리를 한다.

LEA_core.c 코드는 메모리 관리를 위해 malloc( )과 free( )를 사용하여 메모리 할당과 해제를 수행한다. 그런데 C 언어는 메모리를 수동 관리함으로써 메모리 할당이나 해제가 적절히 이루어지지 않으면 메모리나 버퍼 오버 플로우 오류를 일으킬 수 있다.

이에 반해, C++는 객체지향 특성과 풍부한 표준 라이브러리를 제공한다. 물론 코드의 복잡성은 증가할 수 있고 이로 인해 복잡한 데이터 구조 그리고 라이브러 사용으로 오버헤드가 발생할 수 있다. 동적배열 std::vector의 경우, 내부 메모리를 동적 할당함으로써 성능 저하가 일어날 수 있다. 이러한 문제를 해결하기 위해 LEA_core.c와 LEA.cpp 코드에 대한 코드 스멜을 진단하고 품질 개선 작업이 이루어져야 한다.

**표 2. LEA_core.c와 LEA.cpp 코드의 구조**
**Table 2. Code Structure of LEA_core.c and LEA.cpp**

| Items | LEA_core.c | LEA.cpp |
|---|---|---|
| Include | #include 〈stdio.h〉,<br>#include 〈stdlib.h〉 | #include 〈iostream〉,<br>#include 〈vector〉 |
| 전역 정의와 매크로 | 상수, 매크로, 데이터 타입으로 정의 | 클래스 맴버로 정의된 상수와 매크로 정의 |
| 구조와 모듈 | 함수 구성: keySchedule( ),<br>encrypt( ),<br>decrypt( ) | 클래스, 메소드 구성:<br>class LEA { public:<br>keySchedule( ), encrypt( ),<br>decrypt( ) }; |
| 객체 지향 기능 | 상속, 다형성 등 기능 없음 | 클래스, 상속, 다형성 등 기능 있음 |
| 메모리 관리 | 수동 메모리 관리: malloc( ),<br>free( ) 사용 | 자동 메모리 관리: C++ 표준 라이브러리 기능 사용 |
| 표준 라이브러리 사용 | 기본 C 표준 라이브러리인<br>stdlib.h, stdio.h 사용 | C++ 표준 라이브러리<br>vector, iostream 사용 |
| 함수 및 메소드 | 키 스케쥴링 함수<br>keySchedule ( )<br>암호화 함수<br>encrypt ( )<br>복호화 함수<br>decrypt ( ) | 키 스케쥴링 메소드<br>keySchedule ( )<br>암호화 메소드 encrypt ( )<br>복호화 메소드 decrypt ( ) |
| 장점 | 코드의 간결성,<br>높은 이식성<br><br>for(i=0; i〈24; i++) {<br>// 간단한 반복문 사용<br>} | 객체 지향 기능<br>// 클래스 정의<br>class LEA {<br>public:<br>void keySchedule (const std::vector〈uint8_t〉& key);<br>void encrypt (const std::vector 〈uint8_t〉&plaintext, std::vector〈uint8_t〉& ciphertext);<br>void decrypt(const std::vector〈uint8_t〉& ciphertext, std::vector〈uint8_t〉& plaintext);<br>private:<br>std::vector〈uint8_t〉 expandedKey;<br>};<br><br>표준 라이브러리, 타입의 안정성<br>#include 〈vector〉<br>#include 〈iostream〉<br>std::vector〈uint8_t〉 data;<br>std::cout 〈〈 "Data size: " 〈〈 data.size() 〈〈 std::endl; |
| 단점 | 수동 메모리 관리, 추상화 부족<br>uint8_t* data = (uint8_t*)malloc(sizeof(uint8_t) * dataSize);<br>if (data == NULL) {<br>// 메모리 할당 실패 처리<br>} | 높은 복잡성, 오버헤드<br><br>std::vector〈uint8_t〉 data;<br>data.push_back(1);<br>data.push_back(2);<br>std::cout 〈〈 "Data size: " 〈〈 data.size() 〈〈 std::endl; |

표 3은 LEC_core.c와 LEA.cpp 코드의 코드 스멜 전과 후를 비교한 것이다.

**표 3. LEA_core.c와 LEA.cpp 코드의 스멜 전후**
**Table 3. Smell b&a of Code in LEA_core.c and LEA.cpp**

| Items | LEA_core.c | | LEA.cpp | |
|---|---|---|---|---|
| | before | after | before | after |
| 매직 넘버 | DWORD delta[4]={0x…, }; | #define DELTA1 0x… #define DELTA2 0x.. | const word32 delta[8][36]= {…}; | const word32 DELTA1=0x… DELTA2=0x..; |
| 긴 메서드 | 긴 함수 LEA_Key, LEA_Enc, LEA_Dec | 작은 함수 EncryptDecryptBlock, GenerateRoundKeys | 긴 메서드 EncryptBlock, DecryptBlock | 작은 메서드 EncryptBlock, DecryptBlock |
| 주석 수준 | 적절한 주석 | 간결한 주석 | 많은 주석 | 간결한 주석 |
| 코드 중복 | LEA_Enc, LEA_Dec 코드 반복 | 코드 중복 제거, 함수로 추출 EncryptrDecryptBlock | 코드 중복 가능 | 코드 중복 제거, 함수로 추출 |
| 조건문 | 복잡성 높음 | 간결화(EncryptDecryptBlock) | 복잡성 높음 | 조건문 간결화 |

여기서 b&a는 코드 스멜의 전과 후의 상태를 말한다. LEA.cpp의 경우, 매직 넘버를 상수로 정의하여 코드를 읽기 용이하게 한다. 매직넘버는 소스 코드 내 특정 의미를 가지지 않는 정수, 문자열 등의 값을 직접 하드코딩한 것을 말한다. 또한 긴 메서드는 EncryptBlock과 DecryptBlock 함수가 길고 복잡하므로 작은 함수로 분리하여 코드를 읽기 쉽게 할 수 있다. LEA_core.c의 경우, DWORD delta[4] = {ox....};와 같이 매직 넘버가 하드코딩 되어 가독성이 떨어진다. 이를 상수로 정의, 곧 #define DELTA1 0x..... 함으로써, 코드를 읽기 쉽게 할 수 있다. 또한 긴 메서드의 경우, LEA_key, LEA_Enc, LEA_Dec 함수를 사용하는데 길고 복잡하다. 이에 긴 메서드를 작은 함수로 분리 할 수 있는데, GenerateRoundKeys, EncryptDecrypt block처럼 처리함으로써 코드를 읽기 쉽게 할 수 있다. 코드 중복의 경우, LEA_Enc, LEA_Dedc 함수 내의 코드의 반복적인 구조를 가지므로, 코드 중복을 함수로 추출하여, 곧 EncryptDecryptBlock처럼 처리함으로써 중복성 문제를 해결할 수 있다.

## Ⅳ. 결 론

본 논문에서는 LEA_core.c와 LEA.cpp를 비교하여 코드 품질과 복잡성에서의 중요한 차이를 밝힌다. 결과적으로 LEA_core.c는 많은 코드 양, 많은 함수, 많은 속성, 증가된 복잡성, 깊은 중첩 수준을 가지고 있어 LEA.cpp 보다 더 복잡하다. 코드 스멜을 해결하는 것은 소프트웨어의 품질을 향상시키는 데 있어 중요한 문제이다.

향후 연구에서는 실시간 코드 스멜 탐지와 리팩토링을 위한 자동화 도구 개발에 중점을 두어, 소프트웨어의 품질을 개선하고자 한다.

## References

[1] Jae-Won Lee, Chi-Ho Lin, “Design and Implementation of Machine Learning System for Fine Dust Anomaly Detection based on Big Data,” Journal of IIBC, Vol. 24, No. 1, pp. 55-58, Feb. 29, 2024.
DOI: http://doi.org/10. 7236/JIIBC.2024.24.1.55

[2] H. Liu, Y. Zhang, V. Saikrishna, Q. Tian, K. Zheng, “Prompt Learning for Multi-Label Code Smell Detection: A Promising Approach,” arXiv.org, pp. 1-22, 2024.
DOI: http://doi.org/10.48550/arXiv.2402.10398.

[3] T. Sharma, M. Kessentini, “QScored: A Large Dataset of Code Smells and Quality Metrics,” The Proceedings of 18th International Conference on Mining Software Repositories (MSR), IEEE/ACM, 2021.
DOI: http://doi.org/10.1109/MSR52588.2021.00080.

[4] M. Agnihotri, A. Chug, “A Systematic Literature Survey of Software Metrics, Code Smells and Refactoring Techniques,” Journal of Information Process System Vol. 16, No. 4, pp. 915-934, 2021.
DOI: http://doi.org/0.3745/ JIPS.04.0184.

[5] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, “Code smell detection by deep direct learning and transfer learning,” Journal of Systems and Software Vol. 176, pp. 1-25, 2021.
DOI: http://doi.org/10.1016/j.jss.2021. 110936.

[6] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. -G. Gueheneuc, E. Aimeur, “SMURF: A SVM-based incremental anti-pattern detection approach,” Proceedings of WCRE 15, Oct., 2012,
DOI: http://doi.org/10.1109/WCRE.2012. 56.

[7] K. Das, S. Yadav, S. Dhal, “Detecting Code Smells using Deep Learning,” The Proceedings of TENCON, IEEE, 17-20, Oct., 2019.
DOI: http://doi.org/10.1109/TENCON.2019.8929628.

[8] https://refactoring.guru/, Jun 17, 2024.

[9] Jeongha Lee, Hoykyung Bahn, “A Study on Improvement of Buffer Cache Performance for File I/O in Deep Learning,” The Journal of The Institute of Internet, Broadcasting and Communication, Vol. 24, No. 2, pp. 93-98, April 2024.
DOI: https://doi.org/10.7236/JIIBC. 2024.24.2.93

[10] Gwangho Kim, Neunghoe Kim, “Adjustment System for Outlier and Missing Value using Data Storage,” The Journal of The Institute of Internet, Broadcasting and Communication, Vol. 23, No. 5, pp. 47-53, Oct. 31, 2023.
DOI: http://doi.org/10.7236/JIIBC. 2023.23.5.47.

[11] Jeong-Moon Choi, Jungwoo Lee, “A Study on Cell-Broadcasting Based Security Authentication System and Business Models,” Journal of the Korea Academia-Industrial cooperation Society, Vol. 22, No. 5, pp. 325-333. 2021.
DOI: http://dx.doi.org/10.5762/KAIS.2021.22.5.325.

[12] Seongjun Kang, Hyunjun Jung, “Ethereum On-chain Phishing Detection Method using Attention-based 1D CNN,” Journal of KIIT, Vol. 22, No. 5, pp. 23-31, May 31, 2024.
DOI: http://dx.doi.org/10.14801/jkiit.2024.22.5.23.

### 저 자 소 개

홍 진 근(정회원)

- 2000년 : 경북대학교 대학원(공학박사)
- ~ 2004년 : 국가보안기술연구소
- 2004년 ~ : 백석대학교 첨단IT학부 (정교수)
- 관심분야 : AI, 빅데이터, 사이버보안

※ 2024년도 백석대학교 교내학술연구비 지원에 의해 수행된 연구임