DID인증기술 - Capstone Design

KECCAK 알고리즘을 이용한
코드 난독화 프로그램 개발

- 지도 교수 : 신 승 수
- 학　　과 : 정보보호학과
- 팀　　명 : 안약

목 차

I. 서론

1. 연구 배경

자바 언어로 작성된 프로그램은 가상머신 환경에서 동작하는 클래스 형태의 실행 파일로 생성된다. 이는 독립적인 바이트코드로 작성되며 다양한 디컴파일러(역컴파일러)의 개발로 인해 거의 원형에 가까운 코드로 복원된다[1]. 또한 디컴파일러로 복원된 원본 코드는 개발자들의 유지보수와 계속되는 업데이트의 편의를 위해 뛰어난 가독성을 가지고 있어[2] 개발자를 제외한 일반인 사용자들도 해당 프로그램을 비교적 쉽게 분석할 수 있다. 이러한 문제점을 이용하여 자바 코드의 재사용 및 수정을 통한 게임 핵, 크랙 버전 생성 등 저작권 침해와 같은 피해를 발생시킬 수 있다[3].

이로 인해 개발자를 제외한 일반인 사용자가 소스 코드를 이해하기 어렵게 하는 코드 난독화 과정이 추가되었으며 이는 역공학 기법 중 직접적인 파일 실행 없이 소스 코드만을 분석하여 프로그램의 작동 원리를 유추하는 정적분석 방법을 어렵게 한다. 예를 들어 변수의 기능을 유추하지 못하도록 변수의 이름을 무작위로 선언하거나 순환문을 재귀 함수로 바꾸는 등의 방법이 존재한다. 하지만 코드 난독화는 정적분석을 지연시킬 수 있을 뿐 불가능하게 만드는 것은 아니기에 보다 효율적으로 소프트웨어를 보호하기 위한 난독화 기법이 필요하다.

2. 연구 목적

기존 난독화 기법에 관한 연구[4]에서는 변수명을 난수로 치환하는 방법을 기술하였다. 하지만 코드의 길이가 비교적 짧고 줄 바꿈과 띄어쓰기가 존재하여 상대적으로 코드의 가독성이 높아 제3 자의 코드 분석을 쉽게 만든다.

본 논문에서는 이러한 문제점을 개선하기 위하여 변수명 추출 후 의사난수를 추가해 KECCAK으로 해시화 하여 정보 수집으로 인한 공격을 예방할 수 있으며, 변수의 이름을 해시값으로 치환하고 줄 바꿈과 들여쓰기를 제거하여 가독성을 떨어뜨리는 방법을 사용해 난독화를 진행한다. 이러한 방법을 통해 프로그램 분석에 드는 시간을 증가시켜 자바로 작성된 프로그램의 악용사례를 줄이고자 KECCAK을 이용한 코드 난독화 방법을 제안하고 프로그램을 개발한다.

II. 관련 연구

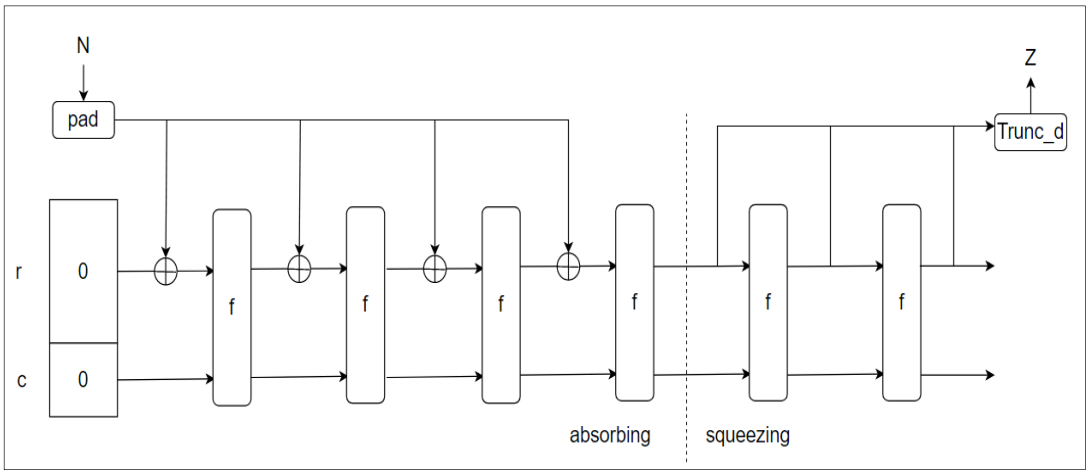
역공학은 프로그램의 동작 원리를 파악하여 해당 프로그램의 복제 또는 오류를 수정할 수 있다. 따라서 역공학에 관한 연구를 바탕으로 프로그램 비교분석을 진행하며 이후 프로그램에서 사용되는 연구들에 대해 기술한다.

1. KECCEK

KECCAK은 스펀지 구조를 가지며, 스펀지 구조는 흡수(absorbing) 과정과 압착(squeezing) 과정으로 이뤄진다.

흡수과정에서는 b 비트 메시지를 r 비트(1,088bit)와 c 비트(512bit)로 나눈다. 그리고 r 비트 값과 패딩값을 XOR 연산하고 그에 대한 결과값과 c 비트를 함수 f에서 24라운드 동안 연산한다. 이후 압착 과정을 통해 r 비트와 c 비트를 함수 f에 넣고 최종 해시값을 출력한다.

이는 역상 저항성, 제2 역상 저항성, 충돌 저항성을 만족하며 수학적으로 증명 가능한 안정성이 보장된다[5]. 동작 과정은 [그림 1]과 같이 진행된다[6].



[그림 1] 스펀지 구조

2. 의사 난수 생성기

의사 난수 생성기(Pseudo Random Number Generator)는 많은 양의 난수가 필요할 때 확률 및 통계를 위해 사용되는 프로그램이다. 초깃값을 이용해 진난수와 가장 가까운 형식의 의사 난수를 생성하며 이러한 의사 난수 생성기에 관한 연구[7]를 바탕으로 무작위 값을 생성하고 변수명에 무작위 값을 추가하여 난독화 실행 시 항상 다른 결괏값을 가질 수 있도록 한다. 이로 인해 정보 수집을 이용한 공격으로부터 안전성을 보장할 수 있다. 따라서 의사 난수 생성기 중 하나인 메르센 트위스터를 기반으로한 random()을 사용하여 추출된 변수에 난수를 추가해 이러한 문제를 해결했다.

3. 난독화

기존 연구에서는 변수명을 난수로 치환하는 난독화를 진행한다. 이러한 난독화 기법은 상대적으로 길이가 짧아 코드 내의 같은 변수명을 찾기 쉽다는 단점이 있으며 줄 바꿈과 띄어쓰기가 존재하여 제3 자의 코드 분석을 쉽게 만든다. 이러한 문제점을 개선하기 위해 변수명을 해시값으로 치환하는 프로그램을 개발하고자 한다.

III. 코드 난독화 프로그램 개발

본 장에서는 자바로 작성된 프로그램을 보호하기 위해 KECCAK을 이용한 코드 난독화 프로그램을 개발한다. 파일을 입력받아 변수명을 추출하여 해시값으로 치환하고 변수 생성 규칙을 준수하기 위해 임의의 값을 생성하여 맨 앞에 추가한다. 이후 줄 바꿈과 들여쓰기를 제거하여 난독화를 하는 과정에 대해 기술한다.

1. 개발 환경

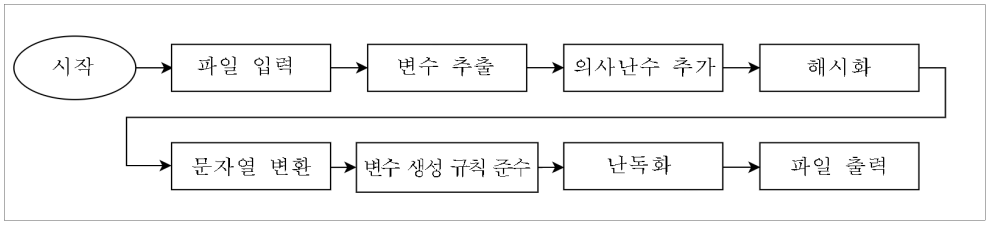
프로그램 동작 환경은 윈도우(Window)를 대상으로 한다. 난독화 프로그래의 개발 언어는 언어는 C++을 사용하며, 개발 플랫폼(Intergrated Development Environment)은 Visual Studio 2022를 사용한다. 자세한 프로그램 개발환경은 <표 1>과 같다.

<표 1> 프로그램 개발 환경

항목	버전
OS	Window 10 64bit
언어	C++
개발 플랫폼	Visual Studio 2022

2. 시스템 동작 과정

사용자가 프로그램을 실행한 후, 코드 난독화를 진행하고 다음 프로그램을 종료하기까지의 시스템 동작 과정은 [그림 2]와 같다.



[그림 2] 시스템 순서도

프로그램은 난독화가 필요한 코드를 txt 형식의 파일로 입력받는다. 이후 사용자가 입력한 파일의 코드를 문자열로 읽어 여러 함수로 전달한다.

입력받은 코드에서 변수를 추출하고 난수 생성기를 통해 임의의 데이터를 변수에 추가하여 난독화된 변수의 이름이 항상 다른 값을 가지도록 변환한다. 그리고 KECCAK을 이용해 입력된 변수명을 해시하여 해시값을 만들어 첫 인덱스에 문자를 추가하여 변수명으로 사용한다. 이후 변수명 생성 규칙을 준수하는 해시값을 문자열로 변환한 후 변수명으로 치환하고 줄 바꿈과 들여쓰기를 제거하여 난독화를 실행하고 완료된 코드를 txt 형식의 파일로 저장하여 사용자에게 제공한다.

3. 프로그램 개발

본 연구에서는 KECCAK을 이용해 변수명과 함수명의 길이를 32자리의 해시값으로 치환하고 맨 앞에 임의의 문자를 추가해 변수명 생성 규칙을 준수한다. 추가로 줄 바꿈과 들여쓰기를 제거하여 가독성을 떨어뜨리는 코드 난독화 프로그램의 개발 하고자 한다.

3.1. 파일 입력

사용자로부터 파일의 경로를 입력받는 함수이다. Get_File 함수를 호출하여 파일을 입력받는다. 해당 함수 내부에서는 FilePath 변수에 입력받은 파일의 경로를 저장하고 fin.open 함수를 사용하여 FilePath 변수에 저장된 경로내의 파일을 읽어온다. 이후, 파일내의 코드를 각각 str 배열과 outstr 배열에 저장한다. 코드는 [그림3]과 같다.

```
string FilePath = Get_File(str, outstr);    // 파일의 경로를 입력받고, 파일의 내용을 문자열로 배열에 저장한다.

string Get_File(string& str, string& outstr) {
    string FilePath;        // 파일의 경로를 입력받는다.
    cout << "파일 경로 입력:";
    cin >> FilePath;
    fstream fin;        // 입력된 파일의 입력 스트림을 생성한다.
    fin.open(FilePath);
    locale::global(locale(".UTF-8"));  // 인코딩 방식을 지정한다.
    char File_Char;

    while (true) {        // 입력된 파일의 코드를 str과 outstr 배열에 저장한다.
        fin.get(File_Char);
        if (fin.fail()) break;
        str.push_back(File_Char);
        outstr.push_back(File_Char);
    }
    fin.close();
    return FilePath;
}
```

[그림 3] Get_File 함수

3.2. 불필요한 자료제거

각각의 str배열과 outstr 배열은 Delete_Symbol 함수에서 주석과 문자열을 제거한다. out str 배열에서 주석표시인 //를 find 함수로 찾아서 주석이 있으면 제거한다. 주석제거 코드는 [그림 4]과 같다.

```
while (true) {   //주석 제거 구현부
    if (outstr.find("//", current) == string::npos) //현재 문자열에서 "//" 이 검색되지 않으면 반복문을 종료
        break;
    current = outstr.find("//", current);   //검색된 문자열("//")의 인덱스를 저장(가장 앞의 것)
    if (outstr[current - 1] != '\"') {      //1-1.검색된 문자열("//") 앞의 문자가 '"' 가 아닐 경우
        while (outstr[current] != '\n') {   //현재 인덱스 값이 가르키는 문자가 '\n' 이 될 때까지 문자를 하나씩 삭제
            outstr.erase(current, 1);   //outsrt 에서 삭제
            str.erase(current, 1);      //str 에서 삭제
        }
    }
    else    // 1-1이 아닌 경우 현재 가르키는 인덱스 값 1 증가
        current++;
}
current = 0;
```

[그림 4] Delete_Symbol 함수(1)

큰따옴표에 들어가는 문자열은 사용자 관점에서 그대로 출력되어야 하기 때문에 난독화가 되면 안 된다. 그러므로 str 배열에서 큰따옴표를 find 함수로 찾은 후 해당 문자열을 제거한다. 문자열 제거 코드는 [그림 5]과 같다.

```
while (true) {  //"" 사이의 문자열 제거 구현부
    if (str.find("\"", current) == string::npos)//현재 문자열에서 큰따옴표가 검색되지 않으면 반복문 종료
        break;
    if (str[str.find("\"", current) - 1] != '\\') {//.1-2검색된 문자(큰따옴표) 앞의 문자가 '\' 가 아닐 경우
        index[0] = str.find("\"", current); //앞의 큰따옴표 위치 저장
        current = str.find("\"", current) + 1;  //현재 위치를 검색된 문자열 위치+1 로 설정

        while (true) {
            if (str.find("\"", current) == string::npos) //현재 위치부터 큰따옴표 가 없을 경우 반복문 종료
                break;
            if (str[str.find("\"", current) - 1] != '\\') {//1-3.검색된 문자(큰따옴표) 앞의 문자가 '\' 가 아닐 경우
                index[1] = str.find("\"", current);//뒤의 큰따옴표 위치 저장
                str.erase(index[0], index[1] - index[0] + 1); //큰따옴표 사이의 문자열 제거 (큰따옴표 포함)
                current = index[0];   //현재 위치를 index[0]으로 설정
                break;
            }
            else//1-3가 아닐경우
                current = str.find("\"", current) + 1;  //현재 값을 찾은 문자 인덱스 +1 로 설정
        }
    }
    else//1-2 가 아닐 경우
        current = str.find("\"", current) + 1;  //현재 값을 찾은 문자 인덱스 +1 로 설정
}
```

[그림 5] Delete_Symbol 함수(2)

3.3. 변수 추출

입력받은 코드에서 find 함수를 이용해 변수명의 자료형을 찾고 그 뒤에 오는 문자부터 특정 조건을 만족하는 문자 전까지를 변수명으로 판단하여 추출한다. 처음에 find 함수가 자료형 char를 찾았으면 입력받은 코드에서 처음부터 끝까지 char를 찾고 char 뒤부터 조건을 만족하는 문자 전까지 나오는 문자열을 추출한다. 이후 int, float, double, void, Scanner, long 순서대로 앞서 설명한 작업을 반복한다. 변수 추출 코드는 [그림 6]과 같다.

```
Find_Variable(fstr, str);          // 난독화를 실행할 변수의 이름을 추출한다.
void Find_Variable(string fstr[], string& str) {

    for (int Variable_Type = 0; Variable_Type < 8; Variable_Type++)
        Save_Variable(str, Variable_Type, fstr);
    fstr[0].push_back(NULL);
    /*fstr[0]에 해당 문자열이 존재 할 경우 삭제한다.*/
    if (fstr[0].find("main;") != string::npos)
        fstr[0].erase(fstr[0].find("main;"), 5);
    while (true) {
        if (fstr[0].find("String[];") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("String[];"), 9);
    }
    while (true) {
        if (fstr[0].find("char;") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("char;"), 5);
    }
    while (true) {
        if (fstr[0].find("String;") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("String;"), 7);
    }
    if (fstr[0].find("void;") != string::npos)
        fstr[0].erase(fstr[0].find("void;"), 5);
    while (true) {
        if (fstr[0].find("int;") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("int;"), 4);
    }
    if (fstr[0].find("float;") != string::npos)
        fstr[0].erase(fstr[0].find("float;"), 6);
    if (fstr[0].find("args;") != string::npos)
        fstr[0].erase(fstr[0].find("args;"), 5);
    if (fstr[0].find("double;") != string::npos)
        fstr[0].erase(fstr[0].find("double;"), 7);
    while (true) {
        if (fstr[0].find("uint32_t*;") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("uint32_t*;"), 10);
    }
    while (true) {
        if (fstr[0].find("uint8_t*;") == string::npos)
            break;
        fstr[0].erase(fstr[0].find("uint8_t*;"), 9);
    }
}
```

[그림 6] Find_Variable 함수

3.4. 의사 난수 추가(생성)

Find_Variable 함수를 호출하여 추출한 각각의 변수에 의사 난수 생성기를 사용하여 난수를 생성하고 해당 난수를 변수명에 추가하여 레인보우 테이블에 대한 공격을 예방하는 함수이다.

Add_Random_Number 함수를 호출하여 변수명 뒤에 생성한 난수를 추가한다. 함수 내부

에서는 random_device 클래스를 사용하여 하드웨어 리소스를 사용하여 예측 불가능한 난수를 생성한다. 이후, mt19937_64 함수를 선언하여 위에서 생성한 난수를 seed값으로 사용하여 64비트 버전의 메르센 트위스터 방식으로 난수를 생성한다.

uniform_int_distribution 클래스를 사용하여 자료형은 int로 난수 생성 범위는 단어마다 난수를 추가해야하기 때문에 영단어 표제수인 0부터 6600000 으로 지정한다[*]. 다음 auto 키워드를 사용하여 생성되는 난수의 타입을 자동으로 결정하고 to_string 함수를 사용하여 생성된 숫자타입의 난수를 int형에서 스트링 형으로 형변환 후에 random_num 변수에 저장한다.

그 후 추출된 변수명이 저장된 fstr 배열 뒤에 난수가 저장된 random_num를 추가하여 ranfstr 배열에 저장한다. 이 과정을 각각의 변수명에 전부 적용한다. 의사 난수 생성 코드는 [그림 7]과 같다.

```
void Add_Random_Number(string* fstr, string* ranfstr) {

    for (int i = 1; fstr[i] != ""; i++) {
        random_device random;                              //하드웨어 리소스를 기반으로 난수를 생성
        mt19937_64 engine(random());                         //메르센 트위스터 방식으로 난수를 생성하겠다는 선언
        uniform_int_distribution<int> distribution(0, 6600000);  //난수의 범위와 자료형 정의. rand() % 100과는 다르게 100도 나올 수 있음
        auto generated = distribution(engine);
        string random_num = to_string(generated);
        ranfstr[i] = fstr[i] + random_num;
    }
}
```

[그림 7] Add_Random_Number 함수

3.5. 해시화

추출된 변수명의 난독화를 위해 변수명을 KECCAK의 입력값으로 넣는다. 해시화 코드는 [그림 8]과 같다.

```
int result = sha3_hash(out, out_length, u8_fstr[d], ranfstr[d + 1].size(), hash_bit);
```

[그림 8] sha3_hash

3.6. 변수 생성 규칙 준수

해시값을 그대로 변수명으로 사용하면 첫 부분이 숫자일 확률이 존재하여 변수 생성 규칙을 위반할 가능성이 있다. 그래서 변수 생성 규칙에 맞게 해시값을 조정해야한다.

number 배열에 순서대로 32개의 16진수 해시값을 모두 더해 저장한다. 이후 0번째 배열부터 mod 91을 해서 값이 65미만으로 나오면 65를 더해주고 더한 값이 90이 넘어가면 39를 뺀다. 다시 number값이 65미만이면 13을 더해줘서 배열에 저장된 데이터를 65에서 90의 범위로 맞춰준다. 이 연산을 number 배열 끝까지 하는데 이때 60부터 90은 아스키코드의 A부터 Z를 나타낸다. 위와 같은 연산을 해서 나온 값을 아스키코드에 대응하는 영문자로 변환 후

변수명 규칙에 맞도록 해시값의 맨 앞에 문자를 추가한다. 코드는 [그림 9]과 같다.

```
/*해시 값 저장, 총합 계산*/
for (int i = 1; i <= out_length; i++) {
    hash_val[d][i] = out[i - 1];
    number[d] += hash_val[d][i];
}

/*총합을 65~89 범위로 재설정*/
number[d] %= 91;
if (number[d] < 65) {
    number[d] += 65;
    if (number[d] > 90) {
        number[d] -= 39;
        if (number[d] < 65)
            number[d] += 13;
    }
}
hash_val[d][0] = (char)number[d];//해시값 맨 앞에 총합을 기반으로 한 알파벳 추가
d++;
```

[그림 9] type_conversion1 함수

3.7. 문자열 변환

변수명을 해시값으로 치환하여 파일에 저장하기 위해 32개의 16진수로 구성된 해시값을 문자열 형식으로 변환하는 함수이다. type_conversion2 함수는 해시값으로 치환된 변수명을 문자열 형식으로 변환한다.

함수 내부에서는 hash_val 배열에 저장된 각각의 16진수 해쉬값을 문자열 형식으로 변환하기 위해 첫 번째 자리에는 해쉬값을 16으로 나눈 후 몫이 10 미만일 경우 문자열형식으로 변경 후 test 배열에 저장하고 해당 몫이 만약 10에서 15사이일 경우 해당 정수에 대응되는 문자로 변환한 후 test 배열에 저장한다. 코드는 [그림 10]과 같다.

```
/*문자열 형태로 변환 과정*/
for (int i = 1; i < 33; i++) {
    if ((hash_val[d][i] / 16) < 10)
        test[d].push_back((hash_val[d][i] / 16) + '0');
    else if (hash_val[d][i] / 16 == 10)
        test[d].push_back('A');
    else if (hash_val[d][i] / 16 == 11)
        test[d].push_back('B');
    else if (hash_val[d][i] / 16 == 12)
        test[d].push_back('C');
    else if (hash_val[d][i] / 16 == 13)
        test[d].push_back('D');
    else if (hash_val[d][i] / 16 == 14)
        test[d].push_back('E');
    else if (hash_val[d][i] / 16 == 15)
        test[d].push_back('F');
```

[그림 10] type_conversion2 함수

두 번째 자리에는 hash_val에 저장된 해쉬값의 나머지를 계산 후 만약 나머지가 10 미만이면 해당 정수를 문자열로 변환 후 test 배열에 저장하고 만약 10에서 15사이일 경우 이에

대응되는 문자를 test 배열에 저장한다. 코드는 [그림 11]과 같다.

```
    if (hash_val[d][i] % 16 < 10)
        test[d].push_back((hash_val[d][i] % 16) + '0');
    else if (hash_val[d][i] % 16 == 10)
        test[d].push_back('A');
    else if (hash_val[d][i] % 16 == 11)
        test[d].push_back('B');
    else if (hash_val[d][i] % 16 == 12)
        test[d].push_back('C');
    else if (hash_val[d][i] % 16 == 13)
        test[d].push_back('D');
    else if (hash_val[d][i] % 16 == 14)
        test[d].push_back('E');
    else if (hash_val[d][i] % 16 == 15)
        test[d].push_back('F');
}
```

[그림 11] type_conversion2

이 과정을 모든 해시값에 적용한다. 코드는 [그림 *]과 같다.

```
/*각각의 해시값에 적용*/
for (int d = 0; d < 255; d++) {
    if (hash_val[d][0] == 205)
        break;

    /**/
    for (int i = 65; i <= 90; i++) {
        if (hash_val[d][0] == i) {
            test[d].push_back((char)i);
            break;
        }
    }
```

[그림 12] type_conversion2

3.8. 난독화

코드 내의 변수명을 해시값으로 치환하고 줄 바꿈과 들여쓰기를 제거하여 난독화를 수행하는 기능을 구현한 함수인 confuse 함수를 호출하여 난독화를 진행한다.

난독화를 하기 전 함수내에서 사용할 변수들을 선언하여 준비과정을 거친다. 코드는 [그림 13]과 같다.

```
confuse(fstr, outstr, String_Hash); //outstr 에 난독화를 실행
void confuse(string fstr[], string& outstr, string test[]) {    //난독화 수행 과정
    int i = 1;
    int current = 0;
    int delindex[500] = { 0, }; //큰따옴표의 위치를 저장하기 위함
    int j = 0;
    bool sw = 0;    //큰따옴표 사이의 문자인지 확인하기 위함

    while (true) {  //큰따옴표 위치값을 추출
        if (outstr.find('₩"', delindex[j] + 1) == string::npos)
        //처음의 경우 첫 위치 부터 큰따옴표를 검색 그 후 직전에 찾은 큰따옴표의 다음 위치부터 검색
            break;

        if (outstr[outstr.find('₩"', delindex[j] + 1) - 1] != '₩₩') {
        //큰따옴표 앞의 문자가 '₩'가 아닐경우 큰따옴표로 인정하여 위치값을 저장
            delindex[j + 1] = outstr.find('₩"', delindex[j] + 1);
        }
        j++;    //저장하고 난 후 다음 칸으로 이동
    }
```

[그림 13] confuse 함수(1)

함수 내부에서는 변수명이 저장된 fstr 배열의 끝까지 반복하는데 큰따옴표 사이의 문자열은 난독화를 하면 안 되기 때문에 bool변수인 sw를 선언하고 큰따옴표 사이의 문자열이 아니면 0을 큰따옴표 사이의 문자열인 경우 1로 판단한다. 큰따옴표 사이의 문자열 판단 코드는 [그림 14]과 같다.

```
while (fstr[i] != "") {

    while (true) {
        if (outstr.find(fstr[i], current + 1) == string::npos)    //난독화 대상의 소스코드에서 변수명을 검색 없을 경우 중지
            break;
        current = outstr.find(fstr[i], current + 1);    //현재 가르키는 인덱스 위치= 문자열을 찾은 첫 위치
        sw = 0; //sw 기본값 0 찾은 문자열이 큰따옴표 사이의 문자일 경우 sw=1
        for (int k = 1; delindex[k] != 0; k += 2) { //큰따옴표 사이의 문자인지 확인하는 과정
            if (delindex[k] < current && current < delindex[k + 1]) {
                current = delindex[k + 1];
                sw = 1;
                break;
            }
        }
```

[그림 14] confuse 함수(2)

파일내의 코드의 sw값이 0이고 outstr 배열이 변수판정을 위한 조건에 부합하면 erase 함수로 해당 문자열을 제거하고 insert 함수로 해당 변수명에 대응하는 해시값으로 치환하는 과정을 모든 변수명에 적용한다. 코드는 [그림 15]과 같다.

```
int j = current + fstr[i].size();
/*변수 판정을 위한 조건들 */
if (sw == 0 && (outstr[current - 1] == '{' || outstr[current - 1] == '[' || outstr[current - 1] == ' ' || outstr[current - 1] == '(' ||
    outstr[current - 1] == ')' || outstr[current - 1] == '=' || outstr[current - 1] == ',' || outstr[current - 1] == '-' ||
    outstr[current - 1] == '+' || outstr[current - 1] == '*' || outstr[current - 1] == '/' || outstr[current - 1] == '₩t'|| outstr[current - 1] == ',' ||
    outstr[current - 1] == '<' || outstr[current - 1] == '>')
    && (outstr[current + fstr[i].size()] == ';' || outstr[j] == ')' || outstr[j] == '(' ||
        outstr[j] == '[' || outstr[j] == ']' || outstr[j] == ' ' || outstr[j] == ',' || outstr[j] == '+' || outstr[j] == '-' || outstr[j] == '*' ||
        outstr[j] == '/' || outstr[j] == '%' || outstr[j] == '>' || outstr[j] == '<' || outstr[j] == ',' || outstr[j] == '.' || outstr[j] == '='))
{
    outstr.erase(current, fstr[i].size());  //변수라고 판정시 변수시작 위치부터 변수이름의 사이즈만큼 outstr문자열에서 지움
    outstr.insert(current, test[i - 1]);    //지운후 변수명과 대응되는 해시값을 입력한다.
    current += 65; //현재 가르키는 위치 값을 해시 값의 사이즈인 65만큼 증가시킨다. **해시값 안을 검사할 필요 없기 때문

    for (int k = 1; delindex[k] != 0; k++) {    //큰따옴표 위치값을 갱신한다. **해시 값으로 치환 후 원래 변수명 뒤에 있던 큰따옴표는 위치값이 달라지기 때문
        if (delindex[k] > current - 65)
            delindex[k] = delindex[k] + 65 - fstr[i].size();
    }
}
```

[그림 15] confuse 함수(3)

그리고 regex_replace 함수를 사용하여 outstr 배열에서 줄 바꿈(\n)과 들여쓰기(\t)를 공백(” “)으로 치환한다. 코드는 [그림 16]과 같다.

```
/*줄바꿈과 들여쓰기 제거*/
outstr = regex_replace(outstr, regex("₩n"), " ");
outstr = regex_replace(outstr, regex("₩t"), " ");
```

[그림 16] confuse 함수(4)

3.9. 파일 출력

난독화된 코드를 원본 코드를 입력받은 txt 파일의 이름에 append 기능을 이용하여‘.ANYAK’을 추가하고 난독화 프로그램이 있는 경로에 txt 파일로 저장한다. 파일을 저장하는 코드는 [그림 17]과 같다.

```
void write_file(string g, string outstr) { //파일 쓰기 g 는 파일 경로
    int index[2] = { 0, };
    index[0] = g.rfind("₩₩");
    index[1] = g.rfind(".txt");
    string gg;
    gg.append(g.substr(index[0] + 1, index[1] - index[0] - 1)); //경로에서 파일 명을 탐색
    gg.append("(obfuscated).txt"); //파일 명에 (obfuscated).txt 를 추가 하여 생성
    std::cout << "Congratulations, obfuscation is complete." << endl;
    std::cout << "Obfuscated File Name: " << gg << endl;
    ofstream fout;
    fout.open(gg); //파일명을 사용하여 파일을 생성
    fout << outstr; //파일 쓰기
    fout.close();//파일 닫기
}
```

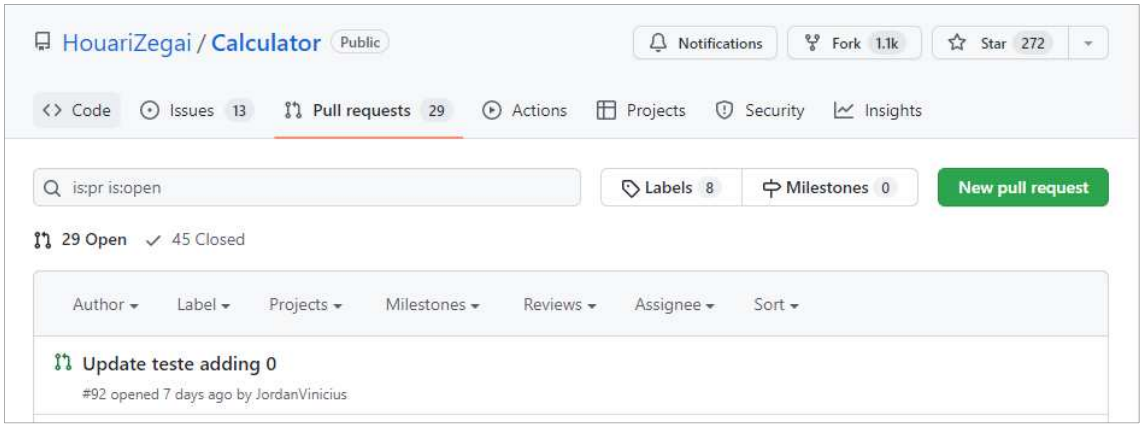
[그림 17] write_file 함수

IV. 분석

본 장에서는 KECCEK을 이용해 기존의 변수명을 난독화 된 변수명으로 치환하고 줄 바꿈과 들여쓰기를 제거하여 원본 코드에 비해 난독화 된 코드가 가독성이 떨어지는지 직접 난독화 하여 비교분석을 진행한다.

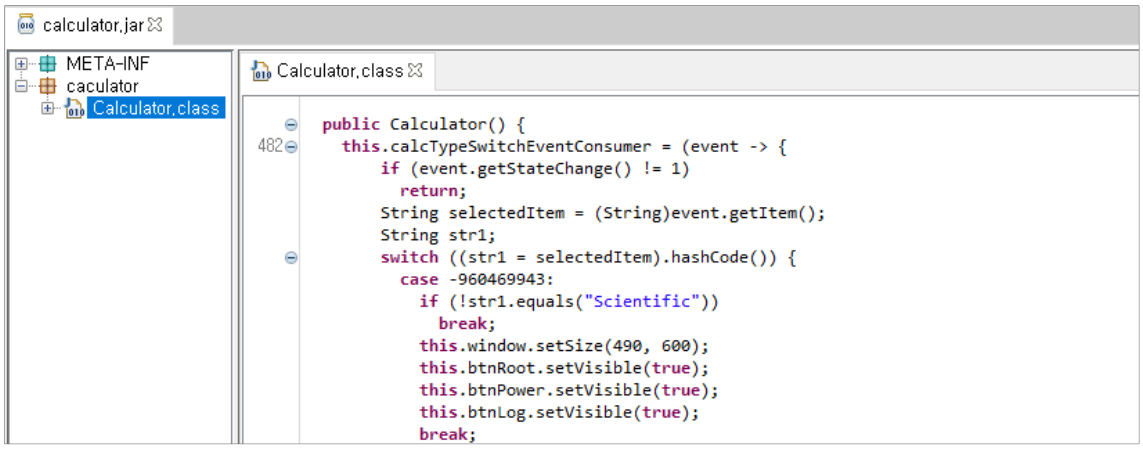
1. 프로그램 실행

자바 난독화 프로그램을 실습하기 위해 오픈 소스 공유 사이트인 깃 허브에서 자바로 작성된 임의의 파일을 다운받는다. 내용은 [그림 18]과 같다[8].



[그림 18] GitHub_Calculator

오픈된 소스로 만들어진 Calculator.jar 파일을 jd-gui 디컴파일러를 이용해 디컴파일 할 경우 원본 소스와 같은 소스 코드를 출력하는 것을 확인할 수 있다. 내용은 [그림 19]과 같다.



[그림 19] jd-gui 디컴파일

디컴파일러로 복원된 원본코드를 본 논문에서 개발한 난독화 프로그램의 입력파일에 맞는 형식으로 변환하기 위하여 텍스트 형식의 파일로 변형한다. 내용은 그림 [20]과 같다.

```
package caculator;

import java.awt.Cursor;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.util.function.Consumer;
import java.util.regex.Pattern;
import java.awt.Color;
import javax.swing.*;
import java.lang.Math;

public class Calculator {

    private static final int WINDOW_WIDTH = 410;
    private static final int WINDOW_HEIGHT = 600;
    private static final int BUTTON_WIDTH = 80;
    private static final int BUTTON_HEIGHT = 70;
    private static final int MARGIN_X = 20;
    private static final int MARGIN_Y = 60;

    private JFrame window; // Main window
    private JComboBox<String> comboCalcType, comboTheme;
    private JTextField inText; // Input
    private JButton btnC, btnBack, btnMod, btnDiv, btnMul, btnSub, btnAdd,
            btn0, btn1, btn2, btn3, btn4, btn5, btn6, btn7, btn8, btn9,
            btnPoint, btnEqual, btnRoot, btnPower, btnLog;

    private char opt = ' '; // Save the operator
    private boolean go = true; // For calculate with Opt != (=)
    private boolean addWrite = true; // Connect numbers in display
    private double val = 0; // Save the value typed for calculation

    /*
        Mx Calculator:
        X = Row
        Y = Column
```

[그림 20] 텍스트 파일 생성

본 논문에서 개발한 프로그램을 사용하여 위에서 만든 텍스트파일에 난독화를 진행한다. 프로그램이 정상적으로 완료될 시 출력되는 화면은 그림 [21]과 같다.

```
파일 경로 입력:C:₩Users₩82105₩Desktop₩비교분석₩before.txt
Congratulations, obfuscation is complete.
Obfuscated File Name: before(obfuscated).txt
계속하려면 아무 키나 누르십시오 . . .
```

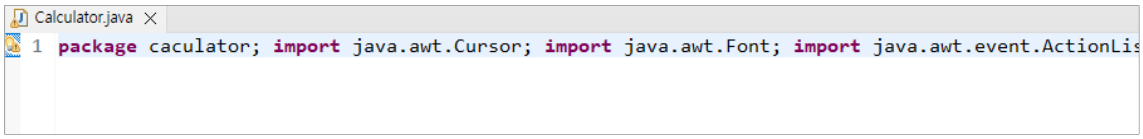
[그림 21] 난독화

난독화를 완료한 파일은 텍스트형식으로 저장되며 해당 파일의 경로에 저장된다. 저장된 파일을 열면 난독화가 된 코드를 확인할 수 있다. 내용은 그림 [22]과 같다.

```
package caculator; import java.awt.Cursor; import java.awt.Font; import java.awt.event.ActionListener; import
java.awt.event.ItemEvent; import java.util.function.Consumer; import java.util.regex.Pattern; import java.awt.Col
import javax.swing.*; import java.lang.Math; public class Calculator { private static final int
L3D8F6579BE2CE59D3CEDA4D2E13FE65FFD78A340C784A93076FCC0C222DC8B33 = 410; private static final int
E7EC4E41555928AA4E4D0294338CFFC8CAE8638684555B5177912143AAECAC0D3 = 600; private static final int
DA3A76E2AB2A86D6173F36EBF9BA44F3106037A0A71D53F5BFF0CEFFE9B35223E = 80; private static final int
P7AECF5A206CA0E7132EEC87BF266226D8DE59E09A1D695691C1CCFC073164258 = 70; private static final int
NC16D10971BABEA08D356BFECA17755D339F30FADE62FB9EFAC8049FAA4E600BB = 20; private static final int
F83840608E908396F1EEFC3EFFB34F05516A61D28F638F4F9BC6EF722E1D6E7AC = 60; private JFrame window; private
JComboBox<String> comboCalcType, comboTheme; private JTextField inText; private JButton btnC, btnBack, btnMod,
btnDiv, btnMul, btnSub, btnAdd, btn0, btn1, btn2, btn3, btn4, btn5, btn6, btn7, btn8, btn9, btnPoint, btnEqual,
btnRoot, btnPower, btnLog; private char U61897BFFFA2AD3DC32EB1D1F12CC12305DA2AE099C6D41FE6E05F48461CA9FF1 = ' ';
private boolean go = true; private boolean addWrite = true; private double
B3FAD3DD7A6EE97B67D291B49B2F92663297FA7239B724B03D0733D12D8C8563A = 0; /* Mx Calculator: X = Row Y = Column
```

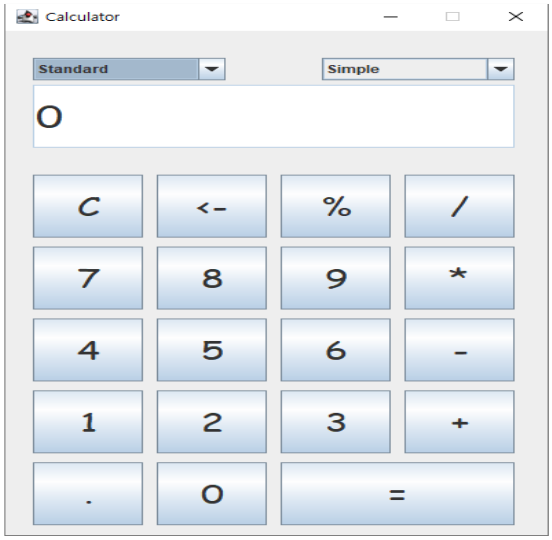
[그림 21] 난독화된 코드

난독화된 코드가 제대로 동작하는지 확인하기 위하여 난독화가 완료된 코드를 자바 컴파일 프로그램(이클립스)에서 실행하는 과정을 거친다. 과정은 그림 [22]과 같다.



[그림 22] 난독화된 코드 사용

컴파일 후 실행되는 화면은 GUI를 지원하는 계산기 프로그램이다. 실행화면은 그림 [23]과 같다.



[그림 23] 실행화면

2. 비교 분석

일반적인 코딩을 하게 되면 각 변수의 이름을 확인하였을때 어떤 역할인지 쉽게 유추 가능하다. 또한, 자바로 작성된 프로그램은 디컴파일러를 사용한다면 원본 코드가 온전히 복원되어 쉽게 수정 및 변조할 수 있다. 그러나 난독화 프로그램을 사용하면 이러한 문제점을 해결할 수 있다. 난독화 전 코드는 [그림 24]과 같다.

```
package caculator;

import java.awt.Cursor;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.util.function.Consumer;
import java.util.regex.Pattern;
import java.awt.Color;
import javax.swing.*;
import java.lang.Math;

public class Calculator {

    private static final int WINDOW_WIDTH = 410;
    private static final int WINDOW_HEIGHT = 600;
    private static final int BUTTON_WIDTH = 80;
    private static final int BUTTON_HEIGHT = 70;
    private static final int MARGIN_X = 20;
    private static final int MARGIN_Y = 60;
```

[그림 24] 난독화 전 변수

난독화 된 코드에서 변수 a, b, c는 각각 변수명 + 난수로 이루어진 값이 KECCAK에 의해 해시값이 만들어지고 일련의 규칙으로 A~Z사이의 문자를 해시값 맨 앞에 붙인 변수명으로 치환되었으며 줄 바꿈과 들여쓰기가 제거되었음을 볼 수 있다.

최종적으로 난독화된 코드의 변수명이 의미없는 값으로 치환되었고 이로인해 변수의 이름을 확인하여 해당 변수의 기능을 유추하기가 매우 힘들다. 또한, 원본 코드에 비해 더 많은 문자가 사용되어 가독성이 훨씬 떨어졌음을 확인 할 수 있다. 내용은 [그림 25]과 같다.

```
package caculator; import java.awt.Cursor; import java.awt.Font; import java.awt.event.ActionListener; import
java.awt.event.ItemEvent; import java.util.function.Consumer; import java.util.regex.Pattern; import java.awt.Col
import javax.swing.*; import java.lang.Math; public class Calculator { private static final int
L3D8F6579BE2CE59D3CEDA4D2E13FE65FFD78A340C784A93076FCC0C222DC8B33 = 410; private static final int
E7EC4E41555928AA4E4D0294338CFFC8CAE8638684555B5177912143AAECAC0D3 = 600; private static final int
DA3A76E2AB2A86D6173F36EBF9BA44F3106037A0A71D53F5BFF0CEFFE9B35223E = 80; private static final int
P7AECF5A206CA0E7132EEC87BF266226D8DE59E09A1D695691C1CCFC073164258 = 70; private static final int
NC16D10971BABEA08D356BFECA17755D339F30FADE62FB9EFAC8049FAA4E600BB = 20; private static final int
F83840608E908396F1EEFC3EFFB34F05516A61D28F638F4F9BC6EF722E1D6E7AC = 60; private JFrame window; private
JComboBox<String> comboCalType, comboTheme; private JTextField inText; private JButton btnC, btnBack, btnMod,
```

[그림 25] 난독화된 변수

3. 한계점

본 논문에서 만든 프로그램으로 난독화한 코드를 디컴파일에 적용하였다. 그러나 프로그램 내에서 줄 바꿈과 공백을 제거하였지만, 디컴파일러에서 자동으로 줄 바꿈을 적용하기 때문에 해당 기능은 구현이 어렵다고 판단되었다. 내용은 [그림 26]과 같다.

Calculator.class

```
import java.awt.event.ItemEvent;
import java.util.function.Consumer;
import java.util.regex.Pattern;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class Calculator {
  private static final int L3D8F6579BE2CE59D3CEDA4D2E13FE65FFD78A340C784A93076FCC0C222DC8B33 = 410;

  private static final int E7EC4E41555928AA4E4D0294338CFFC8CAE8638684555B5177912143AAECAC0D3 = 600;

  private static final int DA3A76E2AB2A86D6173F36EBF9BA44F3106037A0A71D53F5BFF0CEFFE9B35223E = 80;

  private static final int P7AECF5A206CA0E7132EEC87BF266226D8DE59E09A1D695691C1CCFC073164258 = 70;

  private static final int NC16D10971BABEA08D356BFECA17755D339F30FADE62FB9EFAC8049FAA4E600BB = 20;

  private static final int F83840608E908396F1EEFC3EFFB34F05516A61D28F638F4F9BC6EF722E1D6E7AC = 60;

  private JFrame window;

  private JComboBox<String> comboCalcType;

  private JComboBox<String> comboTheme;
```

[그림 26] 자동 줄 바꿈 화면

V. 결론

본 논문에서는 기존의 난독화 기법의 분석을 기반으로 하여, KECCAK을 이용한 코드 난독화 프로그램을 개발하였다.

난독화 과정에서 KECCAK을 사용하기 때문에 역상 저항성에 강하지만 프로그램을 지속적으로 돌려서 정보를 수집한다면 정보 수집에 의한 공격에 취약할 것이라 판단된다. 따라서 이런 공격을 방지하고자 입력된 파일 내의 변수명을 추출하는 과정에서 메르센 트위스터 알고리즘을 사용해 난독화를 수행할 때 마다 중복되는 값이 나오지 않도록 변수명에 난수를 추가하여 해시함수를 사용하기 때문에 역상 저항성을 온전히 보장한다. 그리고 해시값을 역산하여 기존의 변수명을 계산할 수 없게 하였으며 변수명 선언 규칙을 준수하기 위해 해시값을 더하고 연산과정을 거쳐 A~Z까지의 아스키코드 값을 만들어 다시 16진수로 변환하여 해시값 맨 앞에 추가하였다.

이후 코드 내에서 주석과 줄 바꿈이 존재한다면 코드 가독성에 도움을 줄 것이라 생각하여 주석과 줄 바꿈 또한 제거함으로써 최종적으로 위 과정들을 통해 코드를 난독화 하였다. 하지만 이 과정에서 디컴파일러는 코드내의 줄바꿈은 반영하지 않으며 자동으로 줄바꿈을 추가한다는 사실을 알게되어 이 기능은 구현하지 못하였다.

이후 나머지 기능들을 사용하여 코드 난독화가 제대로 수행되는지에 대해 검증하고자 자바로 작성된 프로그램에 대한 난독화 전후의 코드를 디컴파일러를 통해 비교 분석한 결과 본 연구에서 개발된 프로그램을 이용해 난독화한 코드는 변수명이 해시값으로 치환되어 기존 변수명에 대응하는 해시값을 찾기가 매우힘들며, 프로그램 실행시마다 변수명앞에 다른 난수값이 추가되어 악의적인 사용자의 정보수집을 방해하였다. 이를 통해 난독화된 코드는 가독성을 떨어뜨려 악의적인 사용자의 프로그램 무단 복제, 게임핵 제작 등과 같은 악용사례를 저해할 수 있을것이라 기대할 수 있다.

또한, 추후에 코드 내의 띄어쓰기도 제거하는 기능을 추가한다면 더욱더 완벽한 코드 난독화가 가능할 것으로 기대된다.

참고자료

[1] 정효란 외 1명, "효율적인 안드로이드 코드 난독화 기법", 한국컴퓨터정보학회 하계학술대회 논문집 제22권 제2호, 2014.07, pp 57

[2] 손윤식 외 1명, "자바 시큐어 코딩", 정보과학회지, 2010, pp 60

[3] kcopa.or.kr/lay1/bbs/S1T11C293/A/66/list.do?rows=10&cpage=1&cat=&article_seq=&condition=A.TITLE&keyword=저작권+침해

[4] 이주혁 외 1명, "리플렉션과 문자열 암호화를 이용한 안드로이드 API 난독화 도구", 정보처리학회논문지/컴퓨터 및 통신 시스템 제4권 제1호, 2015.01, pp 23

[5] 임경환 외 3명, "자마린으로 개발된 안드로이드 앱의 정적 분석 연구", 정보보호학회논문지, 2018.06, pp 643

[6] KISA, “SHA-3 해시함수 알고리즘에 대한 소스코드 활용 매뉴얼”, 한국인터넷진흥원, 2020,02, pp 1~2

[7] 조영석 외 4명, ”난수 발생기의 비교“, 한국데이터정보과학학지 제3권 제2호, 1992.12, pp 75

[8] https://github.com/HouariZegai/Calculator