동형암호 활용 연구

동형암호를 활용한 비밀번호 유출 감지 프로그램

2020111983 전동원

목 차

- **동형암호 및 Microsoft SEAL 학습**

동형암호는 비대칭 키 암호화 방식의 한 종류이고, 형태가 같은 암호라는 뜻으로 암호화된 상태에서도 덧셈, 뺄셈, 곱셈과 같은 연산이 가능하게 하는 암호화 방식이다.

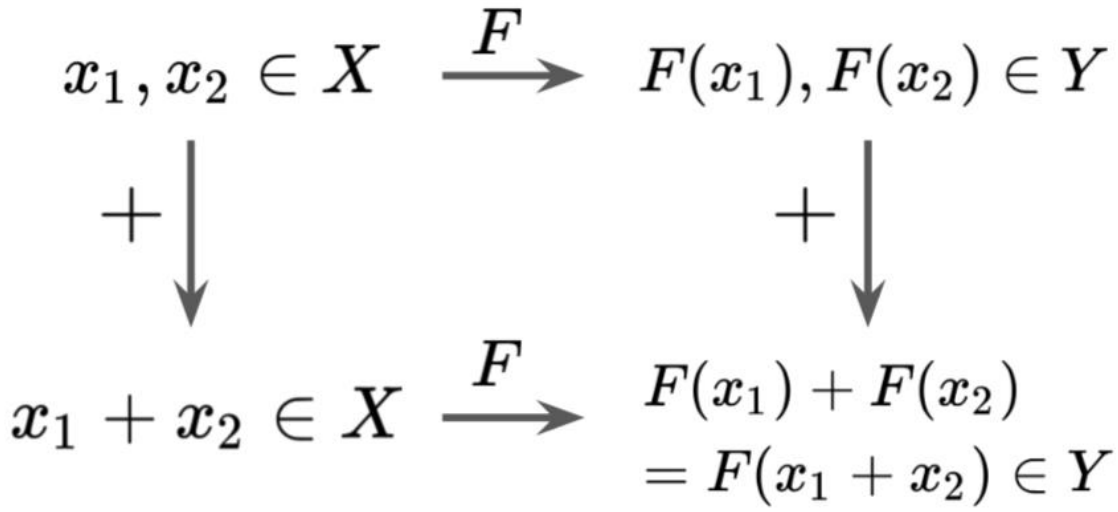


그림 1

위 **그림 1** 처럼 암호화 상태에서 연산한 것이 평문에서 연산한 것과 같은 효과를 보인다는 점에서 같은 형태라는 것이다. 이러한 특성을 이용해서 평문을 보여주지 않고 데이터를 처리할 수 있다.

Microsoft 에서 SEAL 이라는 라이브러리로 위의 동형암호를 활용할 수 있게 하였고 이를 학습했다. 정수 암호화를 다루는 BFV, 실수 암호화를 다루는 CKKS, 비트 연산을 다루는 BGV 방식이 있다. 암호화 파라미터를 조정함에 따라 암호화 정도가 달라진다. 암호화의 과정은 보통 데이터를 벡터의 형태로 저장 -> 평문으로 인코딩 -> 공개 키를 사용해서 암호화 -> 이후 필요한 연산 순으로 진행된다.

암호화 상태에서 연산이 가능하다고는 하나, 무한정 가능한 것은 아니다. 노이즈 예산(Noise budget)은 연산이 가능한 정도를 의미하며, 이게 너무 낮아지면 연산이 되지 않거나 복호화 과정에서 오류가 발생하였다. 이러한 노이즈 예산을 관리하기 위해 재선형화 키가 존재하고, 곱셈 같은 비용이 큰 연산을 진행한 후에 재선형화를 해주면 암호문의 크기도 줄일 수 있고, 노이즈도 효과적으로 관리할 수 있다. 그러나 재선형화는 암호문의 레벨을 감소시킨다. 암호문의 레벨이 모두 감소해버리면 더 이상의 연산이 불가능하게 되기 때문에 곱셈이나 다항식 계산 같은 비용이 큰 연산 이후에만 수행하는 것이 중요하다.

재선형화 말고도 모듈러스 스위칭이라는 것도 존재하는데, 이 기술 또한 재선형화처럼 노이즈를 관리하고 암호문의 크기를 감소시킨다. 당장의 노이즈 예산이 더 감소하더라도 앞으로의 연산을 효율적으로 만든다. 또한 서로 맞지 않는 레벨의 암호문을 같은 레벨로 스위칭하는 역할도 수행할 수 있다. 위 두 가지 기술은 BGV 나 CKKS 처럼 연산 강도가 높은 방식에서 유용하게 사용된다.

본 프로그램을 구현하려면 암호문을 클라이언트에서 서버로 전송해야 하는데, 암호문은 복잡한 구조이기 때문에 그대로 전송할 시 오류가 발생할 가능성이 매우 높다. 따라서 암호문을 직렬화하여 stringstream(바이트 스트림)의 형태로 저장한 뒤 보내야 한다. 직렬화하지 않으면 데이터 손실이나 변조를 보장할 수 없다.

- **주제 선정**

동형암호를 활용하는 주제로 맞으려면 일단 유출되지 않아야 할 중요한 정보를 다뤄야 하고, 이 정보가 서버 혹은 데이터베이스에서 복호화되지 않아야 한다는 조건이 있었다. 기존에 주어진 패스워드 유출 감지 프로그램에 더해 설문 조사를 집계하는 프로그램, 신용 점수로 대출 여부를 판단하는 프로그램, 챗봇 같은 주제를 생각했다. 설문 조사 집계나 신용 점수는 패스워드 유출 감지와 차이가 없어서 제외했다.

챗봇은 가능할지 생각을 해보았는데 자연어 분석은 사실상 불가능하고 가능한 방식은 두 가지 정도로 나뉘는데, 각 챗봇은 다루는 분야가 있으므로 몇 가지 혹은 몇십 가지의 단어를 정해놓고 단어가 출현하는지나 횟수를 측정해 기존에 설정한 문장 후보 중에 골라내서 문장을 분석하는 방법이 있다. 다른 방법으로는 중요한 정보(ex: 개인정보)를 입력할 때만 동형암호로 작동하는 방법이다. 전자의 경우 챗봇을 쓰는 사용자는 보통 즉각적인 반응을 요구하는데, 위와 같이 구현할 때 시간이 매우 오래 걸리거나 클라이언트에 엄청나게 많은 암호문이 도달하기 문제가 될 수 있다. 후자는 앞 문단에서 작성한 설문 조사 집계나 신용 점수 프로그램과 차이가 없다.

따라서 주제는 패스워드 유출 감지 프로그램으로 선정하였다. Microsoft 의 Password Monitor 와 비슷한 기능을 수행하는 프로그램을 만듦으로써 동형암호를 익히는 것이 프로그램 구현의 궁극적 목표이다.

패스워드 유출 감지 프로그램의 동작 순서는 다음과 같을 것이다.

클라이언트에서 암호 입력 -> 서버의 공개키로 암호화 -> 소켓 통신으로 서버로 전송 -> 서버에서 암호문을 읽은 뒤 DB 에 있는 암호화된 비밀번호들을 불러오기 -> 불러온 비밀번호들과 암호문을 빼기 연산 -> 결과를 적절히 연산해서 클라이언트에서 암호문 전송 -> 클라이언트가 복호화 후 유출됐는지 확인하기

위와 같은 방식으로 진행한다면 데이터베이스에는 비밀번호를 확인하지 않고 암호문만 저장하게 되고, 서버에서 클라이언트가 입력한 암호를 바로 DB 의 암호문에서 연산하기 때문에 클라이언트의 암호를 직접 보는 건 아무도 없게 된다.

원래는 뺄셈 연산이 안되는 줄 알고 BGV 의 AND 연산이나, XOR 연산으로 비교하려고 했다. AND 연산을 하면 입력한 비밀번호와 같은 비밀번호가 출력으로 나올 것이고, XOR 연산도 마찬가지로 목적을 달성할 수 있을 거로 생각했다. 하지만 서버에서는 평문을 볼 수 없기에 비트 연산으로 확인하려면 서버에서 연산한 암호문을 다시 클라이언트로 전송해 확인해야 하고, 데이터베이스에 매우 많은 비밀번호의 비교 결과를 모두 클라이언트로 다시 전송하기에는 오랜 시간이 걸릴 것으로 생각돼서 뺄셈이 가능한지 좀 더 알아보았다.

실제로 동형암호에서는 덧셈과 곱셈만 이루어진다. 하지만 모듈러 연산을 사용해서 모듈러 수 p 내에서 연산하기 때문에 뺄셈도 수행할 수 있다. 예를 들어 p 가 11 일 때 10 – 3 을 수행한다면 (p-1)*3 mod p = 30 mod 11 = 8 이 사실상 -3 의 역할을 해주기 때문에 10 에다 더한 후 11 로 모듈러 연산하면 올바른 결과가 나오게 된다.

- **프로그램 구현**

```
hsjw26227
runfj7618
zfbipd41749
hiifx892
tnruj8709
rvcmue0051
mvuh10884
xbhna322
bwrc2164
njewdd707
fans5047
…
```

다음과 같은 비밀번호 파일이 있고 이를 데이터베이스에 저장하였다(약 1500 개)

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
encrypted_pw	mediumblob	NO		NULL	
pw	varchar(100)	YES		NULL	

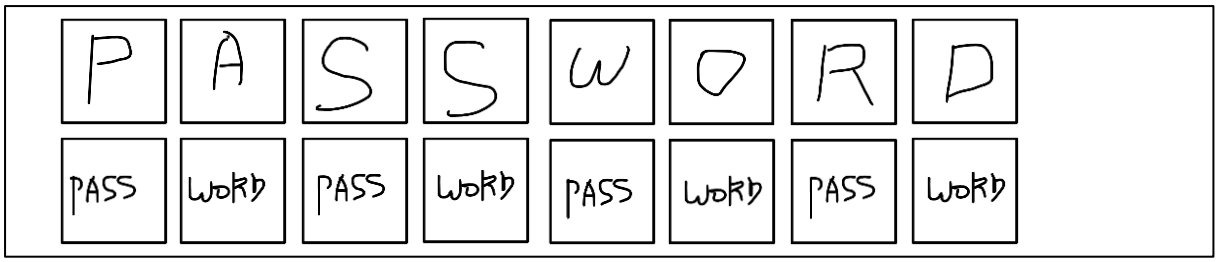
그림 2 - DB TABLE

데이터베이스 테이블 구조는 다음과 같다. pw 는 디버깅용이고 크게 상관은 없다.

```
params.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 40));
```

다음과 같은 코드로 인해 p 가 40bit 근처의 소수로 설정된다. 암호문을 연산할 때 40bit 근처의 소수로 모듈러 연산이 함께 진행되는 것이다.

원래는 비밀번호를 그냥 벡터에 담아서 DB 에 저장할 생각이었는데, 40bit 소수로 연산하기 때문에 한 번에 여러 문자를 담을 수 있다고 한다. 보통 비밀번호는 영어에 아라비아 숫자, 특수문자 등으로 이루어지기 때문에 대부분 8bit 숫자로 하나의 문자를 표현할 수 있다. 소수 p 가 40bit 기 때문에 넉넉하게 32bit, 즉 4 개의 문자까지는 하나의 벡터 항에 넣어서 계산할 수 있다는 것이다. 어차피 같은 비밀번호라면 4 개를 묶어도 뺐을 때 0 이 나올 거라서 문제가 없다. 밑의 그림처럼 묶으면 원래는 1 개 담을 때 4 개씩 담을 수 있으니 매우 좋다.



```
// 4byte 씩 비밀번호를 묶어서 저장(한 비밀번호당 4 개의 항을 차지할 것이다)
vector<uint32_t> read_file(string& file_name) {
	// Read file
	ifstream file(file_name);
	vector<uint32_t> passwords;
	string line;
	if (file.is_open()) {
		while (getline(file, line)) {
			if (!line.empty()) {
				if (line.length() < PADDING_LENGTH) { // password padding
					size_t padding = PADDING_LENGTH-line.length();
					line += string(padding, 0);
				}
				else if (line.length() > PADDING_LENGTH) {
					line = line.substr(0, PADDING_LENGTH);
				}

				for (size_t i = 0; i < line.length(); i += 4) {
					uint32_t part = 0;
					for (size_t j = 0; j < 4; j++) {
						if (i + j < line.length()) {
							part |=
static_cast<uint32_t>(line[i + j]) << (8 * j);
						}
					}
					passwords.emplace_back(part);
				}
			}
		}
		cout << "File Opening... Success.\n";
		file.close();
	}
	else {
		cerr << "File Open Error! (Password File)\n";
	}
	return passwords;
}
```

위 코드는 미리 준비한 비밀번호 리스트를 읽는 함수이다. 쉽게 하려고 16 자로 비밀번호를 PADDING 한다. 비밀번호 뒤에 ₩0(NULL)문자를 패딩해 효과적으로 처리할 수 있다. 이후 4byte 씩 묶어서 uint32_t 벡터에 할당한다.

이후 반환된 passwords 를 slot_count 만큼 할당하여 암호화한 후 DB 에 저장하였다.

이제 클라이언트와 서버를 설명하겠다. 미팅을 통해 새로운 아이디어를 얻어 인코딩 방식을 조작함으로써 오버헤드를 줄일 수 있다는 사실을 알아냈다. 비밀번호를 하나의 암호문으로 보내는 형식이 아니고 묶는 만큼 한 암호문을 꽉 채워서 여러 개를 보내면 DB 에서 곱셈 연산 횟수를 줄이면서 검증할 수 있다고 한다. 서술한 두 가지 방법을 모두 구현하여서 실행 시간을 측정하겠다. 클라이언트는 최종적으로 도착한 암호문을 복호화하는 역할밖에 없기에 서버의 실행 시간을 측정함으로써 비교할 수 있다.

① **일반적으로 비밀번호 전체를 하나의 암호문으로 암호화**

```
EncryptionParameters params(scheme_type::bfv);
size_t poly_modulus_degree = 8192;
params.set_poly_modulus_degree(poly_modulus_degree);
params.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
params.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 40));
```

암호화 계수는 일단 일반적으로 다음과 같이 설정하였다.

위에, DB 에 저장된 형태처럼 클라이언트도 4byte 씩 묶어서 하나의 벡터 항으로 할당한 다음 서버로 보낸다. 이때 아래 코드와 같이 모든 벡터 항에 같은 비밀번호로 쭉 복사해서 암호화하였다.

```
// 암호문 크기만큼 비밀번호를 채워서 보냄
vector<uint64_t> input_password_vec(slot_count, 0);
for (size_t i = 0; i < row_size; i+=4) {
          for (size_t j = 0; j < 4; j++) {
                    input_password_vec[i+j]=input_password[j];
          }
}
```

이후 클라이언트는 서버가 계산한 결과를 받아서 비밀번호가 유출되었는지 확인한다.

다음으로는 서버의 코드이다. 서버는 검증하지 않고 연산만 진행하여 암호문을 압축한다.

```
// DB 에서 꺼내온 뒤 받은 암호문을 빼고 제곱하는 코드
istream* bstream = res->getBlob("encrypted_pw");

Ciphertext db_encrypted_pw;
db_encrypted_pw.load(context, *bstream);

Ciphertext sub_result;
evaluator.sub(db_encrypted_pw, received_pw, sub_result);

Ciphertext squared_result;
evaluator.square(sub_result, squared_result);
evaluator.relinearize_inplace(squared_result, rk);
evaluator.mod_switch_to_next_inplace(squared_result);
```

서버는 클라이언트로부터 받은 암호문을 DB 에서 가져온 암호문에서 뺀다. DB 에 저장된 암호문은 위에서 설명한 것처럼 여러 비밀번호가 쭉 나열된 형태일 텐데, 만약 같은 비밀번호가 있으면 서로 뺐을 때 원본 벡터는 다음과 같을 것이다.

```
X X X X X X X X 0 0 0 0 … (X 는 전부 다른 수)
```

클라이언트는 이렇게 네 개가 연속이면 유출되었다고 판단할 수 있다. 하지만, 이 상태로 클라이언트에 바로 보내기에는 너무 복잡하다. 따라서 정보를 압축해야 한다. 이 네 개의 0 의 정보를 하나의 항에 담기 위해서 좌회전 연산을 한 후 더하기 연산을 세 번 진행하면 맨 앞 항에 네 개의 정보가 모두 담길 것이다. 밑은 회전 후 덧셈 연산을 하는 코드이다.

```
// 회전 후 덧셈 연산을 통해 비밀번호 한 개의 정보를 하나의 항에 담는다.
size_t rotate_size = PADDING_LENGTH / 4;

Ciphertext rotated_result = squared_result;
Ciphertext temp_result = squared_result;
for (size_t i = 0; i < rotate_size-1; i++) {
          evaluator.rotate_rows_inplace(temp_result, 1, gk);
          evaluator.add(rotated_result, temp_result, rotated_result);
}
```

만약 연산을 진행하면 다음과 같은 형태일 것이다.

```
X X X X X X X X 0 X X X … (X 는 전부 다른 수)
```

그러면 클라이언트는 0 이 하나만 있어도 비밀번호가 유출되었다는 사실을 알 수 있다. 그런데 암호문을 서로 뺐을 때 다음과 같은 경우에는 또 문제가 생기게 된다.

```
X X X X X X X X 0 -2 0 2 … (X 는 전부 다른 수)
```

이러면 전부 회전 후 더했을 때 실제로 비밀번호가 일치하지 않음에도 첫 요소가 0 이 되어 버린다. 이것 때문에 위의 코드에서 암호문을 뺀 후 제곱하는 것이다. 제곱하면 마이너스 자체가 사라져 버리니 더해서 0 이 되는 경우가 존재할 수 없다. 물론 더했을 때 modulus 계수 p 의 배수가 되어버리면 0 이 나올 수 있다. 그렇지만 이 부분은 새로운 modulus 계수 q 에 대해서도 테스트하는 식으로 진행하면 해결할 수 있다.

또 다음과 같은 상황일 때도 문제가 발생할 수 있다.

```
X X 0 0 0 0 X X … (X 는 전부 다른 수)
```

앞 비밀번호의 뒷부분이 일치하고 뒤 비밀번호의 앞부분이 일치한다면 절반만 일치하는데도 결과가 일치한다고 나올 것이다. 실제로 비밀번호 뒤쪽은 zero-padding 으로 이루어져 있기 때문에 뒤의 비밀번호가 절반 정도만 일치해도 0 이 될 가능성이 높다. 이 부분은 클라이언트에서 0 인지 확인할 때 네 칸 단위로 확인하는 방식으로 해결해도 되고, 실제로 비밀번호의 절반에서 75 퍼센트 정도가 일치하는 것이므로 위험한 비밀번호로 간주하여 유출됐다고 해도 상관없을 것 같다.

이후 클라이언트가 정말로 하나의 벡터 항만 확인해도 유출 정보를 확인할 수 있게끔 하고자 하였다.

```
// 회전 후 압축 연산하는 코드
size_t numofpasswords = row_size / rotate_size; // LOG2(비밀번호 수)만큼 회전 후 곱하면 모든 비밀번호의 정보를 담을 수 있다
size_t numofsteps = log2(numofpasswords);
temp_result = rotated_result;
evaluator.rotate_rows_inplace(temp_result, numofpasswords/2, gk); // 절반만큼 회전 연산
evaluator.multiply_inplace(rotated_result, temp_result); //

evaluator.relinearize_inplace(rotated_result, rk); // 재선형화 및 모듈러스 스위칭
evaluator.mod_switch_to_next_inplace(rotated_result);

encrypted_result = rotated_result;
numofpasswords /= 2;
```

LOG2(벡터 내의 비밀번호 수)-2 회를 절반 회전 후 곱셈 연산하면 암호문의 정보를 압축할 수 있다. 현재 plain_modulus_degree 가 8,192 라 비밀번호가 한 벡터에 1,024 개가 담겨 있기 때문에 Iteration 마다 절반씩 회전 연산 후 곱셈을 진행하면 벡터의 0 번 항만 확인해도 비밀번호가 유출됐는지 아닌지를 알 수 있다. 이렇게 서버에서 적절한 연산을 거친 후에 클라이언트에 보낸 뒤 클라이언트에서 암호문을 검증한다.

- **문제 발생: 노이즈 예산 부족**

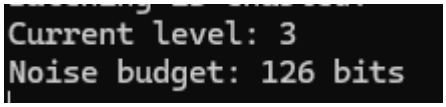
근데 매우 큰 문제가 발생했다. 프로그램을 실행하는 중 계속해서 아래 코드와 같이 연산을 진행한 후 중간중간 남은 노이즈 예산과 레벨을 확인하였다.

```
// 서버에서 디버깅을 위해 노이즈 예산 출력(실제 프로그램에서는 decryptor 는 존재하지 않음)
context_data = context.get_context_data(rotated_result.parms_id());
std::cout << "Current level: " << context_data->chain_index() << "\n";

noise_budget = decryptor.invariant_noise_budget(rotated_result);
std::cout << "Noise budget: " << noise_budget << " bits\n";
```

근데 위의 비밀번호 정보를 압축하는 과정에서 곱셈을 한 번만 진행해도 노이즈 예산이 바닥나는 것을 확인하였다. 노이즈 예산 관리를 위해 연산마다 재선형화와 모듈러스 스위칭을 진행했음에도 불구하고 세 번 이상 곱셈 연산을 수행하지 못했다. 따라서 노이즈 예산을 늘리거나 곱셈 시 노이즈 예산 감소를 줄이게끔 하는 방향으로 코드를 개선하고 실험해 보았다.

◆ **각 암호화 패러미터 값을 수정해보자**

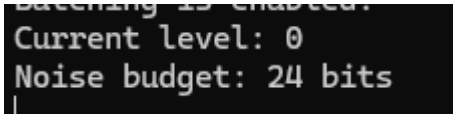


기존 프로그램 구현 방식대로 계수를 설정하였을 때 레벨과 노이즈 예산이다.

- Poly_modulus_degree 값 조정

Poly_modulus_degree 값을 4,096 이나 2,048 까지 줄이면 전체 노이즈 예산은 줄어들 수 있지만 곱셈 한 번당 소모되는 노이즈 양을 줄일 수 있기 때문에 더 많은 곱셈 연산을 수행할 수 있다. 먼저 4,096 으로 줄인 뒤 테스트해 보았다. 클라이언트 단에서 레벨과 노이즈 예산을 확인해 보았다.

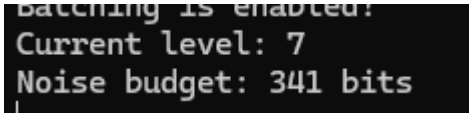
<실행 결과>



생각했던 대로 되지 않았다. 다항식 차수가 낮아서 그런지 노이즈 예산이 시작할 때부터 아예 없어서 프로그램을 진행할 수가 없었다.

반대로 16384 로 올려서 테스트해 보았다.

<실행 결과>

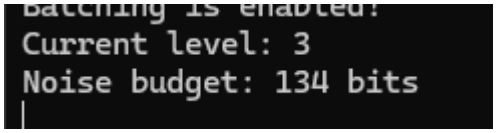


레벨과 노이즈 예산이 충분히 증가하는 것을 확인할 수 있었다. 근데 실행 시간이 너무 오래걸리기도 하고 16,384 로 설정하는 것은 별로 좋지 않다고 하기에 나중으로 미루었다.

- Plain modulus 값 조정

Plain modulus 값을 현재 4 바이트를 하나의 항에 담기 위하여 40bit 로 설정하였는데 이를 32bit 로 설정하고 테스트해보았다. 더 낮추면 4 바이트를 담을 수가 없어서 오류가 발생한다.

<실행 결과>



노이즈 예산이 약간 늘어난 것을 확인할 수 있다. 그러면 오히려 이걸 이용해서 plain modulus 를 20bit 로 줄이고 2 바이트씩 담는 것으로 수정하면 좀 더 효율적인 연산이 가능할 것 같다. 물론 첫 압축과정인 회전 후 덧셈 과정을 진행하는 횟수가 7 회로 늘어나게 되지만, 덧셈 연산은 크게 노이즈를 소모하지 않기 때문에 상관없을 거로 생각한다.

- Coeff modulus 값 조정

조정해보려고 했는데 이게 정해진 계수 모듈이 따로 있는 것 같아서 조정할 수가 없었다.

◆ 연산 과정을 덜 복잡하게끔 해서 노이즈 연산 소모를 줄이기

첫 압축 과정 이후 4 칸마다 2, 3, 4 번째 항은 연산할 필요가 없다. 따라서 밑의 코드처럼 전부 0 으로 처리하면 좀 더 노이즈 소모를 줄일 수 있지 않을까 싶어서 곱해주었다. 바보 같은 생각이었고 노이즈 소모는 똑같이 이루어졌다.

```
// 마스킹을 통해서 필요 없는 정보를 0 으로 만드는 코드(현재 프로그램에서는 삭제됨)
vector<uint64_t> pattern = { 1, 0, 0, 0 };
vector<uint64_t> masked_vec;
for (size_t i = 0; i < row_size/4; ++i) {
          masked_vec.insert(masked_vec.end(), pattern.begin(), pattern.end());
}
Plaintext masked_plain;
encoder.encode(masked_vec, masked_plain);

evaluator.multiply_plain_inplace(rotated_result, masked_plain); // 필요없는 행은 0 으로 처리

evaluator.relinearize_inplace(rotated_result, rk);
evaluator.mod_switch_to_next_inplace(rotated_result);
```

다른 방법으로는 중간에 클라이언트에 한 번 보내고 복호화 후 재 암호화하는 방법도 있을 것이다. 근데 이거는 편법에 가깝고 문제 해결과는 거리가 멀기 때문에 하지 않았다.

◆ 최종 결과

위에서 실험해 본 결과에 따라 poly_modulus_degree 를 16,384, plain_modulus 를 20 으로 수정한 뒤 비밀번호를 2 바이트씩 묶는 것으로 수정하였다. 밑의 코드는 서버에서 수정된 코드이다.

```
// 암호문을 회전 연산 후 곱하여 정보를 압축하는 코드
size_t numofpasswords = row_size; // LOG2(비밀번호 수)만큼 회전 후 곱하면 모든 비밀번호의 정보를 담을 수 있다
while (numofpasswords > rotate_size*2) {
          temp_result = rotated_result;
          evaluator.rotate_rows_inplace(temp_result, numofpasswords / 2, gk);
          evaluator.multiply_inplace(rotated_result, temp_result);
          std::cout << "After Multiplying(Compression)\n";
          evaluator.relinearize_inplace(rotated_result, rk);
          numofpasswords /= 2;
}
encrypted_result = rotated_result;
```

모듈러스 스위칭을 하는 게 오히려 노이즈 예산이 감소하는 것으로 확인되어서 재선형화만 진행하고 모듈러스 스위칭은 마지막에 암호문 전송 시 크기를 줄이기 위해 한 번만 진행하였다. 또한 16,384 로 설정했음에도 불구하고 항 하나에 모든 비밀번호의 정보를 담는 것은 실패하였다. 곱셈

을 10 번은 진행해야 하므로 웬만한 노이즈 예산을 가지고는 수행할 수가 없다. 딱 10 번까지는 곱셈이 잘 진행되었지만 이후 남는 노이즈 예산이 너무 적었다. 5 bit 미만의 작은 노이즈 예산으로는 복호화가 문제가 발생할 수 있기 때문에 곱셈 연산을 한 번 덜 수행하였다.

그러면 이제 암호문 원본 벡터가 O X X X X X X X O X X ... 형태가 될 것인데, 여기서 O 는 각각 512 개의 비밀번호 정보를 담고 있고, X 는 의미 없는 정보이다. 프로그램은 여기까지만 구현하였고, 클라이언트에서는 1 번째, 9 번째 항만 검사해서 유출 정보를 알 수 있다.

◆ **개선할 수 있는 점**

클라이언트가 1 번, 9 번 항만 검사해서 찾을 수 있기에 간단하다고 할 수 있다. 그런데, 클라이언트는 poly_modulus_degree/2 만큼의 벡터를 받기 때문에 나머지 공간은 낭비된다고 할 수 있다. 1 번, 9 번 항을 확인하는 것에 비해 모든 항을 확인하는 것이 큰 실행 시간 차이를 발생시키지 않을 것이다. 이를 이용하면 더 많은 비밀번호 결과를 벡터에 담아 보낼 수 있다.

노이즈 예산을 더 확보하기 위해 최종 암호문에서 1 번 덜 곱셈 연산을 진행한다. 그러면 최종 암호문에는 4 개의 유효한 정보가 담겨있을 것이다. 이를 4 개의 암호문으로 복사한 뒤 000001000... 형태의 평문을 곱하여 각 유효 정보 말고는 존재하지 않는 4 개의 암호문을 만든다. 다음으로 각 암호문을 회전하여 4 개의 항으로 압축한다. 4 개의 항 안에 1,024 개의 비밀번호가 있는 것이므로, 이 과정을 반복해서 모든 벡터를 채운다면 클라이언트에 보내는 한 암호문에 최대 2,048*1,024 = 2,097,152 개의 비밀번호 정보가 담길 수 있다.

② **인코딩 방식을 바꿔 각 글자마다 암호화해서 검사**

1 번 방식을 작성하기 전에 말했던 것처럼 인코딩 방식을 다르게 하여 수행 시간을 단축할 수 있다. 각 암호문을 항 별로 쪼개서 묶어서 DB 에 저장하고, 클라이언트가 서버에게 암호문을 보낼 때도 항 수만큼 암호문을 만들어서 보내는 방식이다.

전체적인 프로그램 구현은 1 번 과정과 비슷하게 진행된다. DB 삽입 코드나 클라이언트 코드는 인코딩 과정을 분리하는 것 외에는 큰 차이가 없다. 클라이언트에서는 8 개의 암호문을 보내야 하는데, 이것이 순서대로 보내는 게 보장되는지도 중요하다. 마침, 사용하고 있는 Winsock 라이브러리는 TCP 통신이기 때문에 패킷의 순서를 보장한다는 장점이 있었다. 패킷을 보낼 때 패킷의 크기만 잘 전달한다면 문제없이 순서대로 보내줄 수 있다.

서버에서는 클라이언트로부터 암호문들을 받아서 DB 의 암호문들에 뺀다. 그러면 같은 비밀번호라면 같은 자리들이 전부 0 이 될 것이다. 이 뺀 암호문들을 전부 더해서 0 이 되면 비밀번호가 일치한다는 사실을 알 수 있다.

사실 이렇게 더하고 또 압축하는 과정을 진행해야 해서, 전체적인 연산 횟수는 1 번과 크게 다르지 않을 것이다. 하지만 1 번과 달리 회전 연산 횟수가 줄어들기 때문에 유의미한 결과 차이를 기대한다. 둘의 올바른 비교를 위해 1,024 개의 비밀번호만 넣어서 비교하겠다.

```
// DB 에서 8 개의 암호문을 가져오는 코드(DB 의 encrypted_pw 가 8 개의 열로 이루어져 있음)
istream* bstream1 = res->getBlob("encrypted_pw1");
istream* bstream2 = res->getBlob("encrypted_pw2");
istream* bstream3 = res->getBlob("encrypted_pw3");
```

```
istream* bstream4 = res->getBlob("encrypted_pw4");
istream* bstream5 = res->getBlob("encrypted_pw5");
istream* bstream6 = res->getBlob("encrypted_pw6");
istream* bstream7 = res->getBlob("encrypted_pw7");
istream* bstream8 = res->getBlob("encrypted_pw8");

Ciphertext db_encrypted_pw1, db_encrypted_pw2, db_encrypted_pw3, db_encrypted_pw4,
db_encrypted_pw5, db_encrypted_pw6, db_encrypted_pw7, db_encrypted_pw8;
db_encrypted_pw1.load(context, *bstream1);
db_encrypted_pw2.load(context, *bstream2);
db_encrypted_pw3.load(context, *bstream3);
db_encrypted_pw4.load(context, *bstream4);
db_encrypted_pw5.load(context, *bstream5);
db_encrypted_pw6.load(context, *bstream6);
db_encrypted_pw7.load(context, *bstream7);
db_encrypted_pw8.load(context, *bstream8);
```

위 코드처럼 클라이언트에서 보낸 8 개의 암호문을 전부 받아서 처리한다.

```
// 2 번 프로그램, 8 개의 암호문을 받아서 8 번 빼고 제곱한다(제곱은 절댓값 처리용)
Ciphertext sub_result1, sub_result2, sub_result3, sub_result4, sub_result5, sub_result6,
sub_result7, sub_result8;
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result1);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result2);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result3);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result4);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result5);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result6);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result7);
evaluator.sub(db_encrypted_pw1, received_pw1, sub_result8);

std::cout << "After Subtraction\n";

auto context_data = context.get_context_data(sub_result1.parms_id());
std::cout << "Current level: " << context_data->chain_index() << "\n";

auto noise_budget = decryptor.invariant_noise_budget(sub_result1);
std::cout << "Noise budget: " << noise_budget << " bits\n";


Ciphertext squared_result1, squared_result2, squared_result3, squared_result4, squared_result5,
squared_result6, squared_result7, squared_result8;
evaluator.square(sub_result1, squared_result1); // 서로 빼고 나서 음수랑 양수가 만나 0 이 되는 경우를
방지하기 위해 제곱한다
evaluator.square(sub_result2, squared_result2);
evaluator.square(sub_result3, squared_result3);
evaluator.square(sub_result4, squared_result4);
evaluator.square(sub_result5, squared_result5);
evaluator.square(sub_result6, squared_result6);
evaluator.square(sub_result7, squared_result7);
evaluator.square(sub_result8, squared_result8);
```

이렇게 다 빼고 제곱도 똑같이 진행해 주었다. 이후 압축 과정은 1 번 프로그램과 진행 과정이 같다. 결과적으로는 4 개의 항에 1,024 개의 비밀번호 정보가 담기게 된다. 1 번 프로그램과 마찬가지로 1,024 * 2,048 개의 비밀번호 정보를 하나의 암호문에 담아서 클라이언트로 보내줄 수 있다.

- **프로그램 분석**

위와 같이 1 번, 2 번 프로그램을 설정하고 두 프로그램의 실행 시간을 비교해보았다. 둘 다 맨 첫 번째 비밀번호를 입력해주었다.

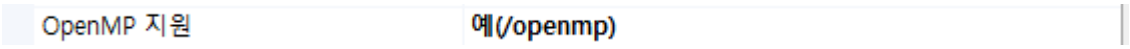
- 1 번 프로그램의 실행 시간

```
Received all data
Ciphertext receive... Complete.
Your password is leaked!
Total Duration of Program Execution: 46.3778
```

- 2 번 프로그램의 실행 시간

```
Received all data
Ciphertext receive... Complete.
Your password is leaked!
Total Duration of Program Execution: 64.8067
```

현재 전체적으로 실행 시간이 오래 걸리긴 하지만, 2 번 프로그램이 더 빨리 끝날 줄 알았는데 훨씬 오래 걸린 것이 예상과 달랐다. 생각해 보니 2 번 프로그램의 경우 8 개의 암호문을 다 빼고, 8 개의 암호문을 모두 제곱하는 데 오랜 시간이 소요돼서 더 오래 걸린 것 같다. 이 부분을 C++의 omp.h 라이브러리를 사용하여 병렬적으로 처리한 뒤 실행 시간을 비교해보았다.



다음과 같이 Visual Studio 프로젝트 설정에서 OpenMP 를 사용할 수 있게끔 설정했다.

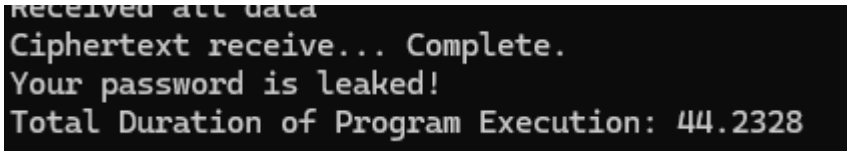
```
// 8 개의 암호문을 빼고 제곱하는 과정을 병렬적으로 처리
#pragma omp parallel for
    for (int i = 0; i < 8; i++) {
        std::cout << "Thread " << omp_get_thread_num() << " is processing square " << i << "\n";
        Ciphertext db_encrypted_pw, sub_result, squared_result;
        if (i == 0) db_encrypted_pw.load(context, *bstream1);
        else if (i == 1) db_encrypted_pw.load(context, *bstream2);
        else if (i == 2) db_encrypted_pw.load(context, *bstream3);
        else if (i == 3) db_encrypted_pw.load(context, *bstream4);
        else if (i == 4) db_encrypted_pw.load(context, *bstream5);
        else if (i == 5) db_encrypted_pw.load(context, *bstream6);
        else if (i == 6) db_encrypted_pw.load(context, *bstream7);
        else if (i == 7) db_encrypted_pw.load(context, *bstream8);

        if (i == 0) evaluator.sub(db_encrypted_pw, received_pw1, sub_result);
        else if (i == 1) evaluator.sub(db_encrypted_pw, received_pw2, sub_result);
        else if (i == 2) evaluator.sub(db_encrypted_pw, received_pw3, sub_result);
        else if (i == 3) evaluator.sub(db_encrypted_pw, received_pw4, sub_result);
        else if (i == 4) evaluator.sub(db_encrypted_pw, received_pw5, sub_result);
        else if (i == 5) evaluator.sub(db_encrypted_pw, received_pw6, sub_result);
        else if (i == 6) evaluator.sub(db_encrypted_pw, received_pw7, sub_result);
        else if (i == 7) evaluator.sub(db_encrypted_pw, received_pw8, sub_result);

        evaluator.square(sub_result, squared_result);
```

다음과 같이 병렬로 처리함으로써 실행 시간 향상을 도모하였다. 병렬 처리에 사용되는 스레드 수는 8 개이다.

- 2 번 프로그램 실행 결과



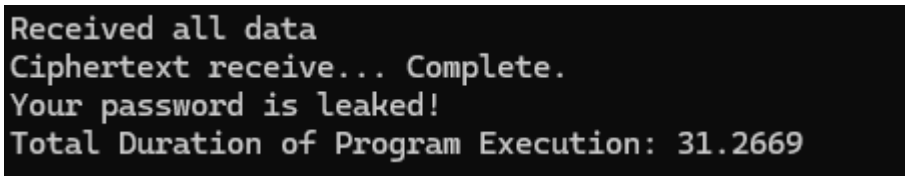
확실히 병렬로 처리하니 매우 빨라진 모습을 볼 수 있다. 근데 1 번 프로그램과 실행 시간에서 별로 차이가 나지 않는다. 1 번 프로그램에서 회전 후 덧셈 연산이 빠진 셈인데, 이렇게 차이가 안 나는 것을 보면 덧셈 연산이 빠지는 게 병렬 처리 오버헤드만큼의 효과를 주지 못하는 것 같다. 사실상 1 번 프로그램과 2 번 프로그램의 실행 시간 차이가 없는 것 같다. 하지만, 2 번 프로그램이 더 많은 비밀번호를 처리할 가능성이 있다. 지금은 1,024 개만 담았기 때문에 만약 8,192 개 전부 담아서 보낸다고 할 때, 노이즈 예산이 충분하다면 훨씬 많이 압축할 수 있다는 장점이 있다. 같은 시간에 거의 10 배가량의 비밀번호를 담을 수 있으므로 2 번 프로그램의 성능이 더 좋다고 할 수 있다.

둘의 비교보다는 일단 실행 시간 자체를 단축해 보겠다. 현재 노이즈 예산 관리를 위해 모듈러스 스위칭을 진행하지 않았는데 제곱했을 때 한 번, 압축 연산할 때 두 번에 한 번씩 모듈러스 스위칭을 진행하고 실행 시간을 측정해 보았다. 노이즈 예산 문제로 곱셈을 한 번 덜 해야 한다.

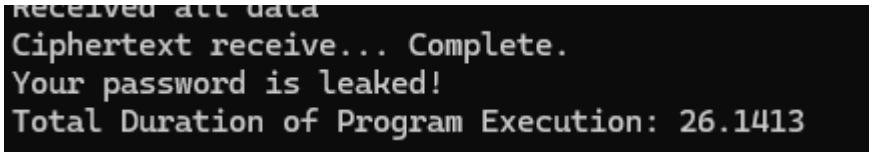
```
if (cnt % 2 == 1) {
                                        evaluator.mod_switch_to_next_inplace(rotated_result);
}
```

cnt 변수를 선언해서 2 번에 한 번씩 모듈러스 스위칭을 하도록 하였다.

- 1 번 프로그램 실행 시간

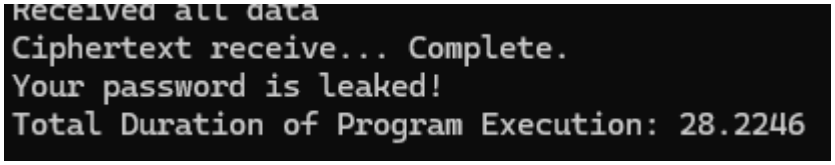


- 2 번 프로그램 실행 시간

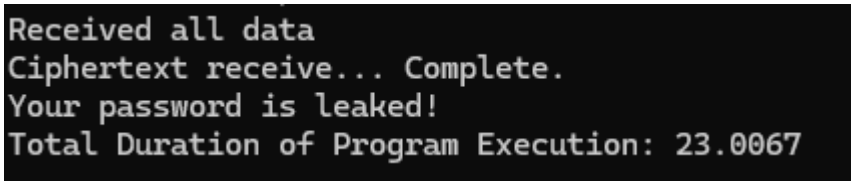


이전보다 실행시간이 많이 단축된 것을 볼 수 있다. 1 번과 2 번 프로그램의 시간 차이도 점점 늘어나고 있다. 모듈러스 스위칭을 세 번에 두 번 하도록 수정한 뒤 실행해 보겠다.

- 1 번 프로그램 실행 시간



- 2 번 프로그램 실행 시간

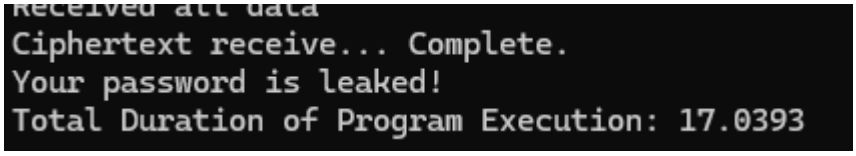


더 많이 단축된 모습을 볼 수 있다. 현재 본인이 구현한 프로그램은 20 초대까지 줄이는 것이 한계이고, 이보다 더 줄이려고 한다면 연산 방식을 변경하거나 곱셈 횟수를 줄여야 할 것 같다. 모듈러스 스위칭을 더 진행하면 노이즈 예산이 바닥나서 오류가 발생한다.

- 결과 및 소감

동형암호를 활용하는 프로그램의 경우 곱셈보단 덧셈 위주로 진행하는 것이 효율적이라는 것을 알 수 있었다. 무엇보다도 노이즈 예산을 잘 관리하기 위해 재선형화를 잘 해주는 것이 중요했다. 재선형화는 암호문의 차수를 낮춰서 이후 연산에서의 노이즈 증가를 줄여주었다. 연산 횟수를 늘리려면 무작정 노이즈만 관리하면 됐는데, 막상 또 노이즈만 관리하다 보면 프로그램 실행 비용이 너무 커졌다. 따라서 모듈러스 스위칭도 적절히 수행해 줘야 프로그램 실행 시간을 단축할 수 있었다. 현재 2 번 프로그램 기준으로 얘기하면 평균 25 정도 걸리는데, 여기에 위에서 말한 것처럼 2 백만 개가량의 비밀번호를 모두 합치더라도 30~35 초 미만으로 걸릴 것이다(마스크 벡터 곱셈 한 번 + 덧셈 여러 번이지만 덧셈은 오래 걸리지 않음).

이렇게 비밀번호 유출을 탐지하는 과정은 오래 걸려도 문제가 되진 않을 것이다. 웹사이트에 로그인하면 사용자가 웹사이트를 사용하다가 30 초나 1 분 뒤에 결과가 나오고, 사용자에게 알림 형식으로 제공하면 되기 때문이다. 그런데 동형암호를 활용하는 프로그램 중에 만약 빠른 반응성이 필요한 프로그램이면 어떡할까? 애플의 스팸 번호 탐지 같은 경우는 전화가 오자마자 최대 5 초 안에는 스팸인지 아닌지 알아내야 하는데 위처럼 프로그램을 제작하면 이미 전화를 한창 진행하고 있을 때 알림이 올 것이다. 그러면 프로그램의 존재 의미가 없다. 그래서 차라리 현재 프로그램처럼 하나의 항으로 정보를 압축하는 것보다, 적당히 압축한 뒤 클라이언트로 여러 개의 암호문으로 나누어 보내주는 것이 실행 시간의 측면에서는 나을 거라고 생각된다. 실제로 클라이언트에게 암호문을 보내는 시간이 더 짧을 것이기에 실행 시간을 줄일 수 있다.



위 사진은 압축 횟수를 3 회로 줄였을 때의 실행 시간이다. 압축 횟수를 줄이면 실행 시간이 매우 단축될 것이다. 만약 프로그램의 결과가 천천히 나와도 상관없다면(비밀번호 탐지) 노이즈 관리에

만 집중하며 최대한 많은 정보를 담고, 반응성이 중요하다면 압축은 최소화하며 암호문을 가볍게 하는 것에 집중하는 것이 좋다고 결론지을 수 있다.

프로젝트를 구현하기 위해 처음 동형암호를 이론적으로 학습할 때는 솔직히 각 함수나 내용 자체가 이해하기 어려웠는데, 막상 직접 구현하면서 실행 시간 차이나 연산 횟수 차이를 비교하고 나니 각 함수의 역할을 자세히 알고, 동형암호에 대한 이해도가 확실히 올라간 것 같다. 막연히 안전하다고만 이해했는데, 실제로 보니 서버도 암호문만 보기 때문에 공격자는 클라이언트를 대신하여 쿼리를 보내서 유출 정보를 얻는 것 외에는 비밀번호에 접근할 방법이 없는 것 같다. 심지어 쿼리도 비정상적으로 많이 보내면 서버에서 차단할 수 있기 때문에 공격자는 방도가 없는 것 같다. 이처럼 매우 안전한 암호화 방식이라는 것을 알 수 있었다. 암호문은 매우 복잡할 텐데 암호화 상태에서 연산이 동작한다는 것이 매우 신기했다. 현재는 4 세대까지 나왔다는데, 앞으로 발전해서 비교 연산 같은 것도 가능하게 된다면 더더욱 많은 분야에 활용할 수 있을 거로 생각한다.