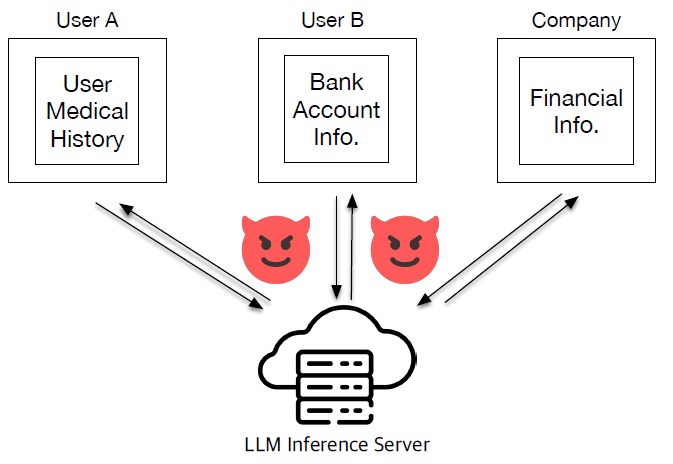
**Introduction**

2017년 “Attention Is All You Need”라는 논문 발표 이후 OpenAI, Google, 그리고 Meta와 같은 수 많은 기업들이 자신만의 거대언어모델(Large Language Model; LLM)을 개발해 출시해오고 있다. LLM의 추론 성능을 향상시키기 위해 다방면으로 수행된 연구들의 결과를 통해 확립된 사실은 추론 성능이 LLM의 모델 파라미터 개수와 비례한다는 점이다. 이에 따라 추론 성능을 향상시키기 위해 LLM의 크기는 나날이 커지고 있는 실정이다.

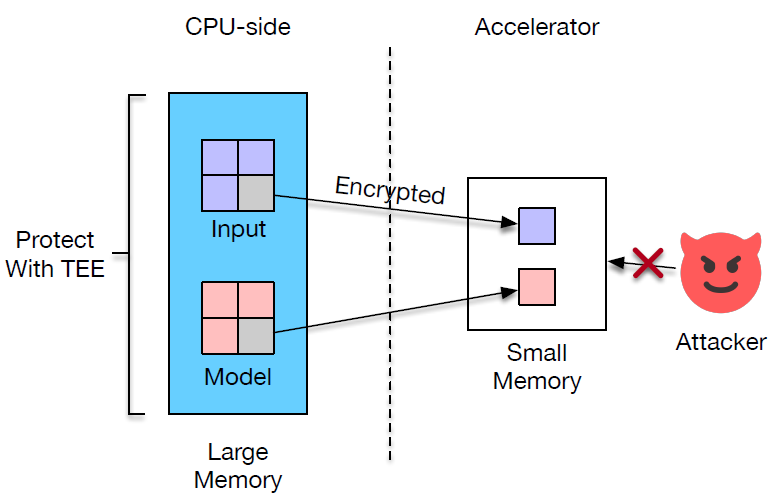
반면에 LLM의 크기가 커진다는 것은 연산장치 및 메모리와 같은 컴퓨팅 자원이 그만큼 더 요구된다는 점이다. 현재 LLM의 크기는 이미 상당히 커져 일반적으로 추론 요청을 하는 사용자의 로컬 컴퓨팅 머신에서는 효율적인 연산이 어렵기에 컴퓨팅 자원이 풍부한 클라우드 서버에 추론 요청을 보내 연산한 뒤 결과물만 돌려받는 방식(Computation offloading model)을 채택한다. 이러한 방식을 통해 LLM 추론 서비스 사용자는 고성능 로컬 컴퓨팅 머신 없이도 서버의 컴퓨팅 자원을 사용해 쾌적한 추론 서비스를 누릴 수 있다.

하지만 위와 같은 Offloading model은 사용자의 민감정보를 포함하고 있는 추론 요청을 공격자가 존재할 수 있는 클라우드 서버에 어떠한 보호장치도 없이 보내기 때문에 보안적인 측면에서는 안전하지 않다. 아래 그림은 개인 사용자나 회사의 민감정보들을 포함한 LLM 추론 요청이 클라우드 서버로 내보내질때 공격자에 의한 민감정보 탈취 가능성을 보여준다.

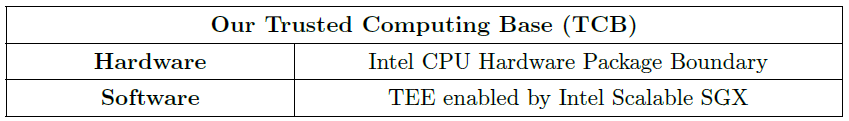
**연구목표 및 필요성:** 정리하자면 매우 거대해진 LLM 추론 연산을 위해서 추론 요청을 클라우드 서버에 내보내 연산 후 결과를 되돌려 받는 오프로딩 방식은 효율적인 측면에서 필수적이다. 반면에 사용자 민감정보가 담긴 요청을 외부로 오프로딩 시키는 것은 공격자로 인한 탈취 및 위변조와 같은 공격으로부터 안전하지 않다. 이에 따라 본 연구의 목표는 추론 요청을 외부 가속기에 오프로딩 시켜 효율적인 추론 연산이 가능하게 하면서도 가능성 있는 공격자로부터 사용자 민감정보를 안전하게 보호할 수 있는 보호 메커니즘 연구를 목표로 한다.

**Background**

**Trusted Computing Base (TCB):** 본 연구에서 타겟하는 연산 모델은 앞서 언급하였듯이 Offloading scheme을 따른다. 이러한 연산 모델에서는 평소에 추론 요청에 필요한 사용자 입력을 공격자로부터 안전한 신뢰 가능한 영역에 가지고 있다가 행렬곱셈 연산과 같은 많은 컴퓨팅 자원을 요구하는 연산 처리시에만 외부 가속기에서 추론 요청을 부분적으로 처리한다.

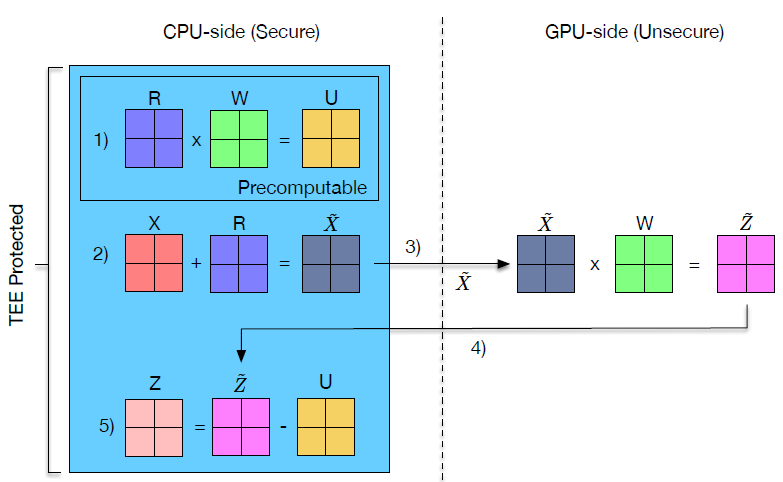
본 연구에서는 TCB를 TEE (Trusted Execution Environment; CPU-side에 안전구역을 형성하여 공격자로부터 Code와 Data를 보호함) 및 TEE에서 동작하는 소프트웨어로 한정한다. 또한 외부 가속기(우리의 예제에서는 GPU가 외부 가속기) 사용시에는 사용자 입력을 적적히 암호화 후 내보내 공격자가 의미를 알 수 없도록 한다. 아래 그림에서는 Offloading model을 시각화 한다. 평소에 추론을 위한 사용자 입력과 인공지능 모델 가중치는 CPU-side에서 안전하게 보호된다. 외부 가속기 사용시에 사용자 입력은 기밀성 보장을 위해 암호화 후 외부로 내보낸다. (\*만약 모델 가중치가 보호의 대상이 아닌 경우에는 TEE에 갖고 있지 않는 것이 성능에 좋음)

**Threat Model:** 바로 앞서 지금까지 설명해온 내용들을 정리하자면 우리의 위협 모델(Threat model)이 아래와 같이 정의된다.



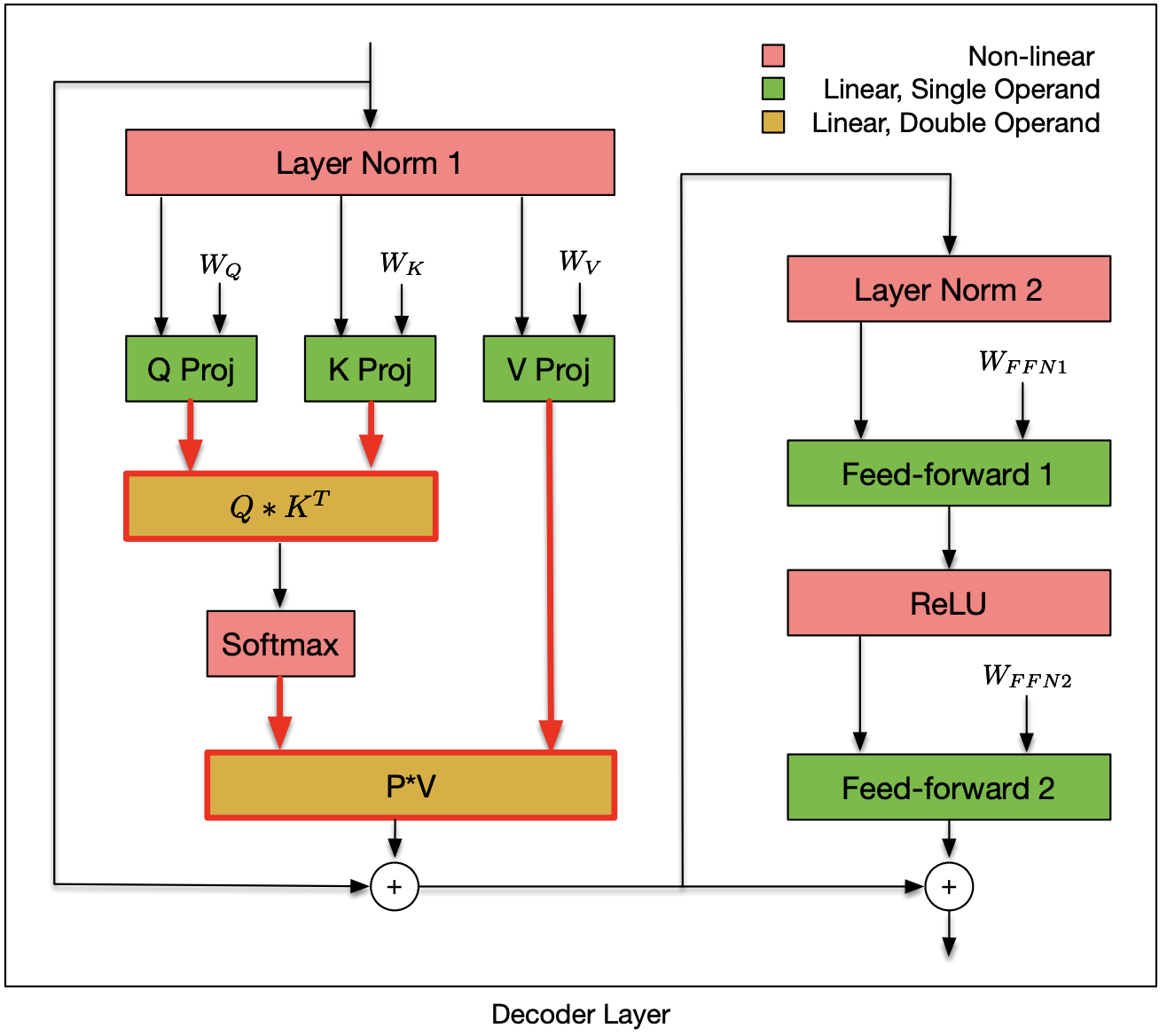
하드웨어 측면에서는 인텔 CPU 하드웨어 패키지 내부는 외부로부터 관측이 불가능하다고 가정한다. 또한 소프트웨어 측면에서는 보호되어야할 대상은 모두 TEE내부에서 안전하게 관리되어지고 있다고 가정한다.

**기존 연구:** 기존 연구는 LLM 이전의 인공지능 모델(CNN, DNN, …)들에 대해 외부 가속기 사용시 기밀성을 보장하는 암호화 기법에 대해 제안한다. 기존의 연구 중 하나인 Slalom (ICLR 2019)에서는 연산을 위해 외부 가속기로 내보낼 때 사용자 입력에 CPRNG을 통해 생성된 난수를 더해 얻은 결과물을 생성해 외부로 유출 시 공격자가 의미를 알아볼 수 없게 감춘다. 예를 들어, 사용자 입력이 X이고 모델 가중치가 W라 할 때 우리가 최종적으로 가속기를 통해 얻고자 하는 결과물을 Z=XW이다. 하지만 기밀성 보장을 Slalom에서는 위해 사용자 입력 X에 암호키 R을 더해 을 내보낸다. 가속기에서는 Z대신 를 돌려보낸다. 원래 구하려면 값 Z를 구하려면 를 해주어야 한다. 다행히도 사용자 입력 암호화를 위한 R은 사전에 미리 선정할 수 있고 모델 가중치인 W도 학습 이후에 고정된 값이기 때문에 복호화를 위한 는 미리 계산해 가지고 있다가 가 돌아오면 단순히 빼주면 된다. Slalom에서 제안한 암호화 기법은 DNN 및 CNN과 같이 사용자 입력과 모델 가중치를 행렬곱하는 Linear layer연산에는 매우 효과적이나 추후 자세히 설명하겠지만 LLM 추론시에는 사용할 수 없게 된다. 아래 그림은 Slalom에서 Linear layer연산을 하는 시각화 한다. 1) 암호화키 R은 사전에 선정하여 모델 가중치 W와 미리 행렬곱셈해두어 복호화키 U를 저장해둔다. 이후 2)에서 사용자 입력 X가 발생하면 미리 선정해둔 R을 더해 로 암호화한다. 3)에서는 외부 가속기에서 를 구한다. 4)에서는 암호화된 상태에서 연산된 결과물을 다시 CPU-side TEE로 되돌려 받는다. 마지막으로 5)에서는 원래 구하고자 하던 값 Z를 구하기 위해 에서 미리 구해 저장해 두었던 U를 뺀다.



**Motivation**

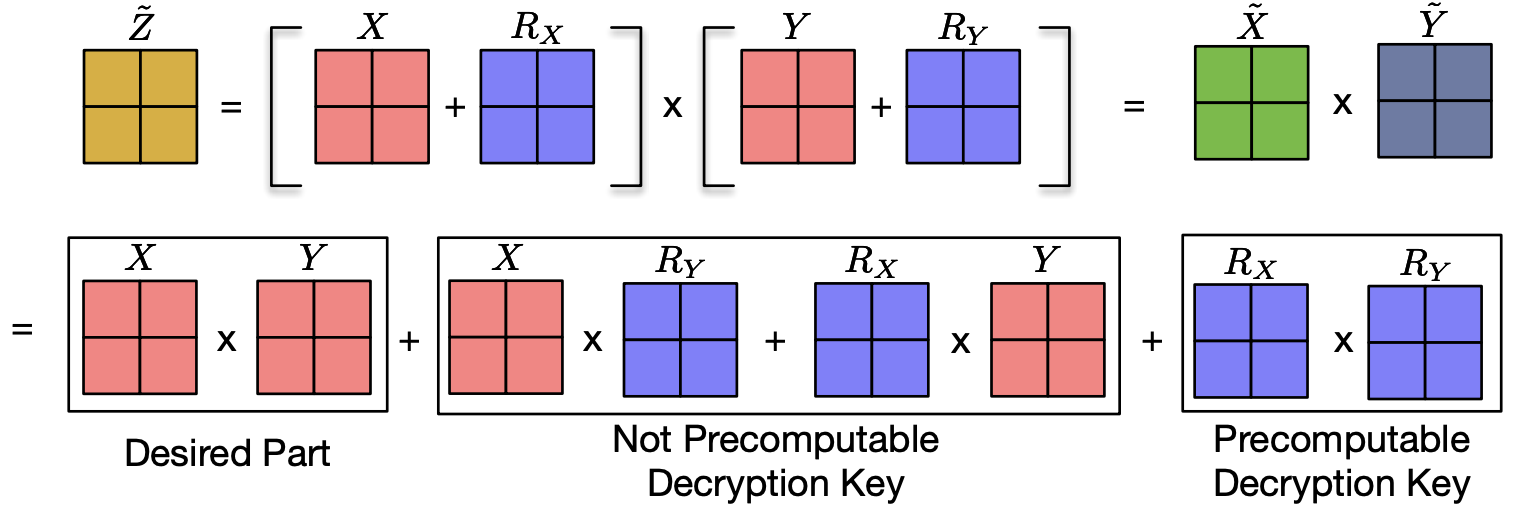
**Attention mechanism의 독특한 연산 특징:** 현재 대부분의 Large Language Model은 Transformer 블록을 기반으로 동작한다. 더 자세히는 트랜스포머 블록에는 Attention mechanism에 의해 동작하는데 이러한 메커니즘은 DNN과 CNN과 같은 기존의 인공지능 모델과는 다른 연산적 특징을 보여준다. 아래는 OPT 모델의 (디코더 부분) 트랜스포머 구조이다.



먼저 위 디코더 트랜스포머 블록에서 빨간색으로 칠해진 연산은 Non-linear 특성을 갖는 연산들이다. 다음으로 초록색으로 칠해진 연산은 Linear layer 연산들이다. Linear layer 연산의 특징은 행렬곱셈을 위한 한쪽의 피연산자는 사용자 입력이고 다른 쪽 피연산자는 모델 가중치이라는 점이다. 마지막으로 노란색으로 표시된 연산(QK^T, PV)들은 행렬곱셈을 위한 양쪽 피연산자 둘 다 사용자 입력에서 도출된다는 점이다. 이렇게 양쪽 피연산자 둘 다 사용자 입력에서 도출된 경우 프라이버시를 보존하며 효율적인 연산을 가능하게 하는 메커니즘을 어렵게 한다.

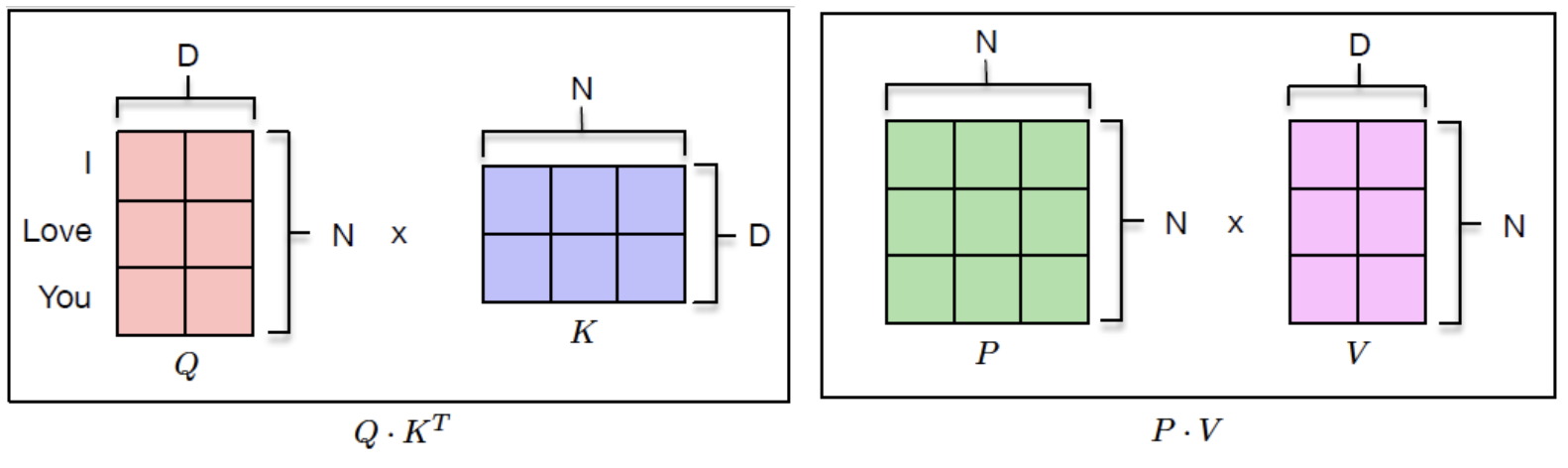
**한쪽 피연산자만 사용자 입력인 경우 프라이버시 보존:** 한쪽 피연산자만 사용자 입력이고 다른쪽은 모델 가중치인 경우에 프라이버시 보존을 하며 효율적인 연산을 제공하는 메커니즘은 앞서 Slalom 기법을 언급했듯이 이미 존재한다. 가속기 사용을 위해 외부로 사용자 입력을 내보내기전 암호키를 각 원소에 더해 공격자가 의미를 알 수 없도록 감춘다. 밖으로 내보내진 암호화된 사용자 입력은 가속기에서 행렬곱셈 후 다시 TEE로 되돌려진다. 되돌려진 암호화된 결과는 원래 구하고자하는 값을 구하기 위해 복호화 키를 빼어낸다. (\*자세한 내용은 Background 참조)

**양쪽 피연산자 둘 다 사용자 입력인 경우 프라이버시 보존:** 양쪽 피연산자 둘다 사용자 입력에서 도출된 경우 앞서 제안된 Slalom방식을 사용할 수 없게 된다. 아래 그림은 Slalom에서 제안한 기법을 naïve하게 적용하였을때의 상황이다.

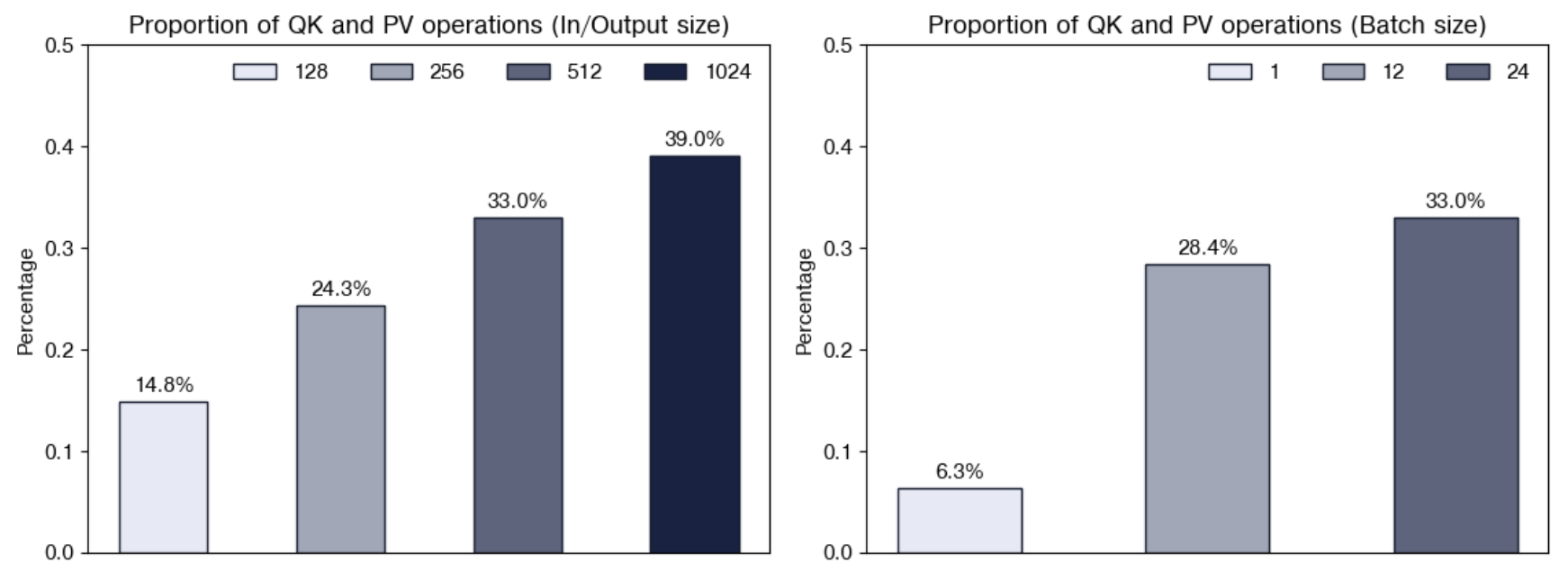


피연산자 X와 Y는 각각 와 를 더해 암호화된 와 를 생성 후 외부 가속기로 내보내고 결과값 를 구한다. 이를 풀어서 다시 써보면 아래와 같다.

위 식에서 우리가 원래 구하고자 하던 값은 Z이다. Z를 구하기 위해서는 를 에서 빼주어야한다. 이때 양쪽 피연산자 둘 다 사용자 입력에서 도출되었을때 Slalom 방식의 비효율성이 드러난다. 다행히 항은 미리 선정하고 미리 구해둘 수 있으나 항은 사용자 입력 X, Y가 알려지기전까지 TEE내에서 미리 구해놓을 수 없다. 미리 구할 수 없다면 X, Y가 알려졌을때가 되어서야 CPU TEE내에서 복호화키를 구하기 위해 두번의 행렬곱셈을 해야하는데 이는 원래 구하고자 하던 값 를 가속기를 사용하지 않고 직접 CPU에서 구하는 것보다 항상 비효율적이다. 즉, Slalom방식은 이러한 상황에서 사용할 수 없다.

**양쪽 피연산자 둘다 사용자 입력에서 도출된 경우 항상 CPU TEE에서 연산하는 방식:** 기존의 Slalom에서 제안한 방식을 사용할 수 없다면 양쪽 피연산자 둘 다 사용자 입력으로부터 도출된 연산인 QK^T와 PV의 경우 그냥 CPU TEE내에서 처리하는 방향을 고려해볼 수 있다. 피연산자 입력의 크기가 작을때는 CPU TEE에서 처리하는 편이 외부 가속기에 내보내 처리하는 것보다 효율적일 수 있다. 하지만 QK^T와 PV의 연산량은 입력 크기에 제곱 비례해서 증가한다는 점이 존재한다. 사용자 입력의 크기가 N일때 QK^T와 PV 연산량 둘 다 이다.

사용자 입력의 크기가 커질수록 (길어질수록) 추론을 위해 드는 전체 시간에서 더 많은 부분을 차지한다는 것을 아래의 실험에서 확인 할 수 있었다.



위 그래프는 end-to-end 추론 시간에서 사용자 입력 크기에 따른 시간 비중이다. 예를 들어 입력과 출력의 크기가 각각 128일때 전체 추론 시간에서 QK^T와 PV연산을 수행하는 데 소요된 시간 비중이 14.8%였으며 입력과 출력의 크기가 각각 1024일때는 39%의 비중을 차지했다. 이는 사용자 입력의 크기와 출력의 크기가 커질 수록 QK^T와 PV의 연산량이 빠르게 증가하며 이는 결국에는 CPU TEE에서 단독으로 처리하는 것이 힘들어 질 것을 예측한다.

**Affine Cipher**

Affine cipher 암호화 기법은 정수론 기반의 monoalphabetic substitution cipher 암호화 기법이다. 알고리즘에 대한 자세한 내용은 링크 참조 바람: <https://en.wikipedia.org/wiki/Affine_cipher>

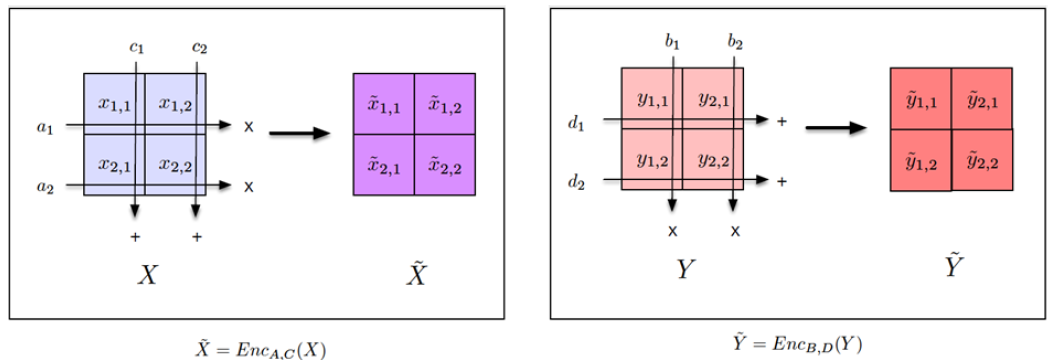
**정수론 기반 암호화 기법 사용을 위한 Quantization**

LLM이 일반적으로 채택하는 데이터 형식은 FP16이다. 하지만 LLM의 크기가 커지면 커질수록 연산량, 저장공간, Host-Device간 데이터 송수신양이 크게 증가함에 따라 다양한 모델 경량화를 위한 노력들이 존재한다. 다양한 경량화 기법 중 하나인 양자화(Quantization)은 데이터를 더 적은 수의 비트로 근사화해 모델의 전체적인 크기를 줄이는 기법이다. 예를들어 Smoothquant [1]에서는 사용자 입력과 모델 가중치를 둘 다 FP16에서 INT8로 양자화해 추론 연산에 드는 저장공간과 데이터 송수신양을 절반으로 줄인다.

본 연구에서 다루는 암호화 기법은 정수론(Number theory)에 기반한다. 따라서 모델 가중치의 원본 데이터 형태인 실수(Floating point)는 사용할 수 없기에 정수(Integer)로 먼저 양자화해주어야 한다. 본 연구에서는 사용자 입력과 모델 가중치 둘다 양자화가 가능한 Smoothquant를 활용해 Llama3 모델을 양자화 한다.

**본 연구에서 제안한 암호화 기법**

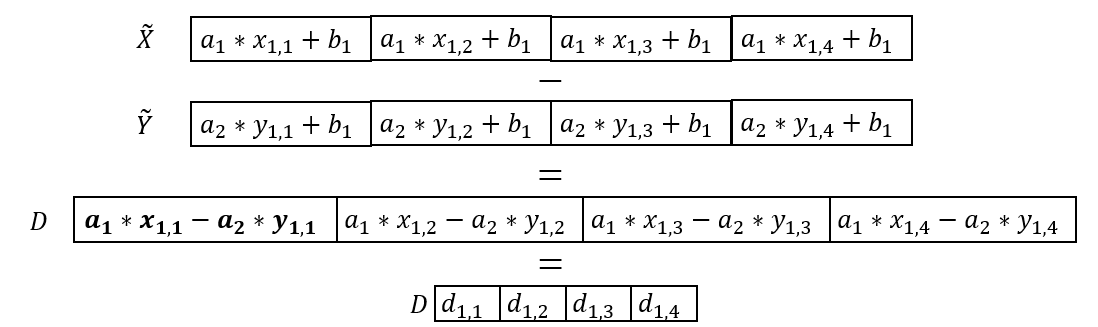
본 연구에서는 앞서 언급하였던 Affine cipher를 기반으로 행렬곱셈의 양쪽 피연산자인 입력 행렬을 각각 암호화하고 암호화된 상태에서 연산이 가능하게 하는 것을 목표하였다. 입력 행렬의 행과 열은 각각 곱셈 암호키와 덧셈 암호키를 공유한다. 암호키를 공유(재사용)하는 것은 암호화된 상태로 가속기로부터 되돌려진 결과를 해독하기 위해 필요한 복호화키 생성을 에서 로 한 차원 효율적이게 한다. 이는 기존에 CPU에서 복호화키 생성에 드는 비용을 획기적으로 줄일 수 있게한다.



하지만, 암호키를 다수의 원소 간에 공유하는 행위는 이미 Cryptography 필드에서는 암호화 시스템에 취약점을 발생시킨다고 결론 내렸다. 즉, 암호키는 재사용하면 안되고 만약 재사용한다면 어떤 형태의 실제 실행가능한 공격 시나리오가 생긴다는 것이다. **결론적으로 본 연구에서 제안한 암호화 기법은 암호키를 재사용하는 것으로 인해 취약점을 드러낸다.**

**본 연구가 제안한 암호화 기법에 대한 가능한 공격 시나리오**

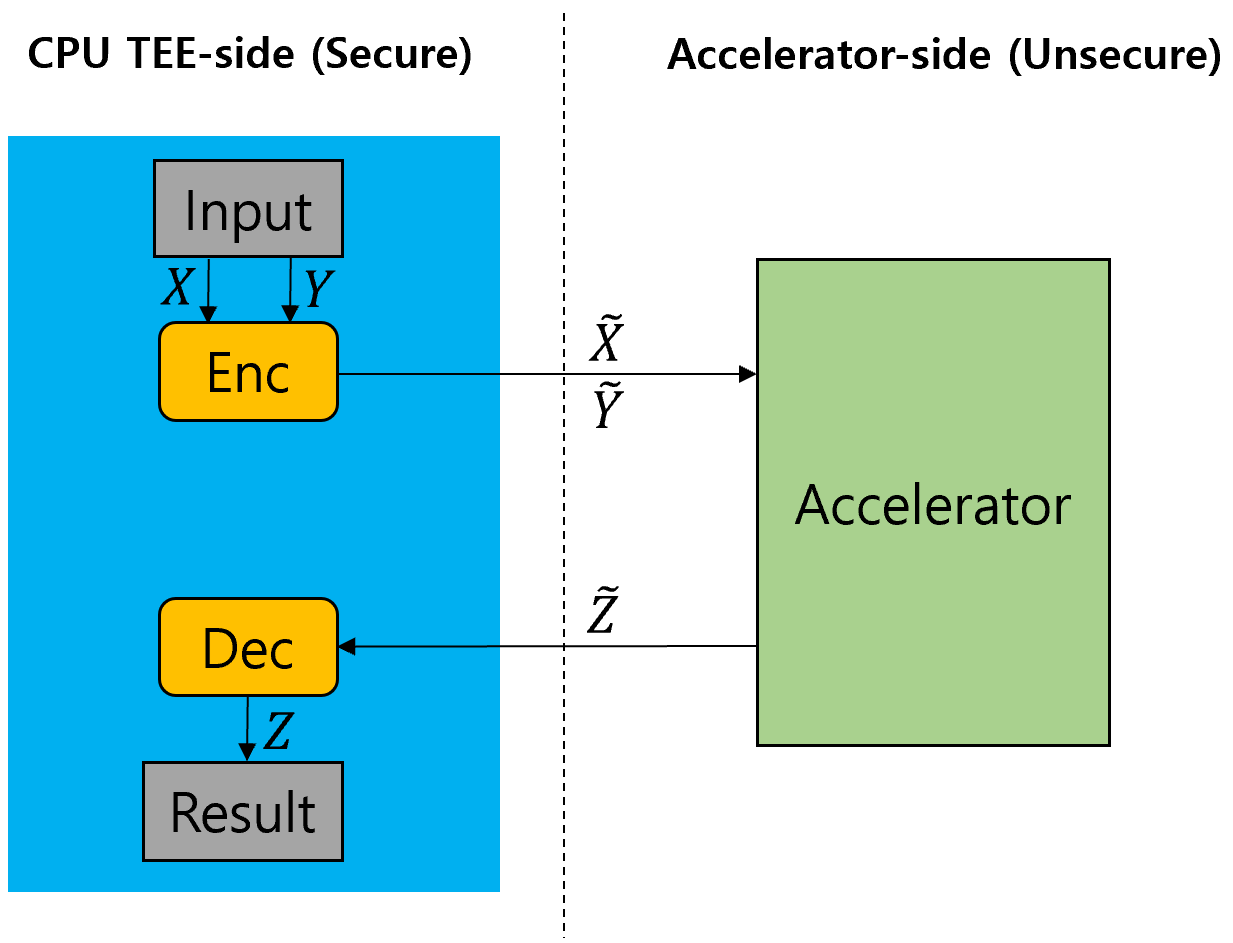
1. 먼저 암호화된 두 토큰을 서로 빼서 차이벡터 D를 구한다. 그러면 토큰 가 공유하고 있던 덧셈 암호키 성분이 제거된다.



1. 차이벡터 D에서 임의로 두개의 값을 선택한다. 예를 들어 을 선택 그러면 아래에서처럼 연립방정식을 세울 수 있다.
2. 위 연립 방정식에서 공격자는 자신이 선정한 X와 Y 그리고 D의 값들을 이미 알고 있는 상태이다. 또한 위 연립방정식의 해는 항상 존재한다 (사용자가 선택한 X와 Y 그리고 암호키를 사용해 d값이 구해졌기때문). 그렇다면 해가 존재하고 연립 방정식 두개가 존재하는 상황에서 암호키 는 손쉽게 찾을 수 있다.
3. 위와 같은 공격 시나리오를 통해 본 연구에서 제안했던 암호화 기법은 뚫리게 된다.

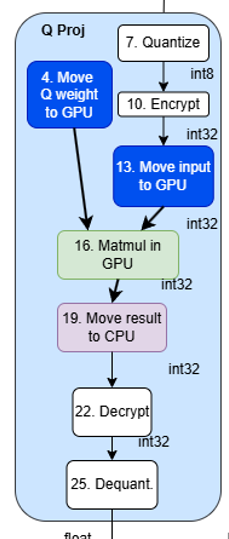
**Computation Flow Model**

우리가 채택하는 연산의 흐름은 아래 그림에서처럼 표현된다. 먼저 사용자 입력(X와 Y)은 CPU TEE에서 존재하고 안전하지 않은 외부 가속기로 보내지기전 특정 암호화 기법에 의해 암호화가 된다(와 로 암호화). 암호화된 입력들은 안전하게 가속기로 보내 결과 를 계산한다. 마지막으로 CPU-TEE로 보내 원래 구하고자 하던 결과 를 복호화 과정을 통해 구해낸다. 본 연구에서 사용하는 CPU TEE는 Intel Scalable SGX이며 사용하는 가속기는 NVIDIA GPU를 활용한다.



**Parallelization Opportunity**

Transformer 아키텍처의 decoder layer는 상당 부분 자연적으로 병렬화가 가능하다. 이러한 병렬화가 된다는 장점을 최대한 살리기 위해 decoder layer의 많은 부분을 독립적인 operation들로 정의하였고 dependency graph에 따라서 올바르게 동작하도록 구성하였다. 아래 그림은 Q projection 연산을 dependency graph로 표현한 것이다. 전체 dependency graph는 doc폴더에 있음.



간단하게 설명을 하자면 7번에서 FP16형태의 사용자 입력을 Smoothquant를 사용하여 INT8로 양자화한다 (CPU TEE 내부). 10번에서는 양자화된 입력에 암호화를 적용한다 (CPU TEE 내부). 13번에서는 암호화가 적용된 사용자 입력을 GPU에서 연산을 위해 TEE에서 GPU로 전송한다. 4번에서는 모델 가중치 Q를 GPU로 전송 시킨다 (Flexgen처럼 모델 가중치를 CPU에 갖고 있다가 필요시에만 GPU로 전송). 16번에서는 사용자 입력과 모델 가중치를 가지고 행렬 곱셈 연산을 한다. 19번에서는 GPU에서 연산된 결과를 CPU로 다시 전송한다. 22번과 25번에서는 각각 GPU로부터 되돌려 받은 결과물을 복호화 및 dequantization한다. 지금까지 언급된 연산들을 독립된 연산으로 dependency graph에 맵핑해보면 4번 연산과 7-10-13번 연산이 병렬화 가능함을 알 수 있다.

**Reference**

[1] https://arxiv.org/pdf/2211.10438