Leaking Secrets in Homomorphic Encryption with Side-Channel Attacks

Furkan Aydin and Aydin Aysu

Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA.

Contributing authors: faydn@ncsu.edu; aaysu@ncsu.edu;

Abstract

Homomorphic encryption (HE) allows computing encrypted data in the ciphertext domain without knowing the encryption key. It is possible, however, to break fully homomorphic encryption (FHE) algorithms by using side channels. This article demonstrates side-channel leakages of the Microsoft SEAL HE library. The proposed attack can steal encryption keys during the key generation phase by abusing the leakage of ternary value assignments that occurs during the number theoretic transform (NTT) algorithm. We propose two attacks, one for `-O0` flag non-optimized code implementation which targets `addition` and `subtraction` operations, and one for `-O3` flag compiler optimization which targets `guard` and `mul_root` operations. In particular, the attacks can steal the secret key coefficients from a *single* power/electromagnetic measurement trace of SEAL's NTT implementation. To achieve high accuracy with a *single-trace*, we develop novel machine-learning side-channel profilers. On an ARM Cortex-M4F processor, our attacks are able to extract secret key coefficients with an accuracy of 98.3% when compiler optimization is disabled, and 98.6% when compiler optimization is enabled. We finally demonstrate that our attack can evade an application of the random delay insertion defense.

Keywords: Homomorphic Encryption, SEAL, Number Theoretic Transform, Compiler Optimizations, Side-Channel Attacks, Machine Learning

1 Introduction

Fully homomorphic encryption (FHE) allows arbitrary computations on encrypted messages without the need for decryption [15]. FHE is useful, e.g., for cloud computing where the untrusted cloud can compute on encrypted data and the user, who holds the secret key, can decrypt the returned result. Therefore, HE preserves the privacy and confidentiality of data while allowing computations in untrusted environments. Although FHE is an evolving approach with mathematically provable security guarantees, their physical implementations can have vulnerabilities. For example, the first successful physical side-channel attack on FHE [3] has recently been demonstrated, revealing the encrypted message by exploiting the side-channel leakage of Gaussian sampling operations.

In this work, we reveal new side-channel vulnerabilities of Microsoft SEAL—an FHE software library [30]. SEAL is a high-profile target that has recently gained significant recognition in the literature and has been used in many applications [4, 12, 22]. Our attack focuses on the number theoretic transform (NTT) function of SEAL executed during the key generation. We first show that the NTT processes ternary values (-1, 0, or +1) that correspond to the secret key coefficients. Then, we build a side-channel attack that can extract this information from NTT operations. The challenge in attacking this stage is being limited to a *single-trace* measurement. We address this challenge by developing a multi-stage neural network based side-channel classifier. Finally, we implement a defense based on random delay insertion for the NTT and assess its effectiveness against our *single-trace* attacks.

In this work, we also analyze the effect of compiler optimizations on the side-channel leakage of SEAL's NTT. In the ARM compiler, there are different optimization levels for the target ARM-M4F processor: `-O0`, `-O1`, `-O2`, and `-O3`. Our first attack focuses on `addition` and `subtraction` operations of the SEAL's NTT code with complied `-O0`, which corresponds to no optimization at all. This setting is commonly used when attempting a constant-time implementation in order to avoid the compiler optimizing the loops and ruining the developer's work [8]. In terms of performance, settings from `-O1` to `-O3` indicate varying degrees of optimization. The setting `-O3` provides the highest level of optimization. Therefore, we also analyze SEAL's NTT implementation with `-O3` optimization level in this work. However, our previously shown side-channel leakages do not exist with `-O3` flag compiler optimization. Despite compiler optimizations eliminating previous side-channel leakages, we demonstrate new side-channels that occur during `guard` and `mul_root` operations in SEAL's NTT implementation.

Our work is different from earlier *single-trace* side-channel attacks on the NTT [12, 20, 25, 27]. The timing leakage analysis by Drucker *et al.* achieves a low success rate of 9% [12] because the attack only focuses on branch executions of butterfly in NTT. This attack is inapplicable to SEAL because its butterfly unit is constant-time. Kim *et al.* propose an ML-based side-channel attack on NTT [20]. The attack exploits Montgomery reduction operation that does not exist in SEAL's NTT. Primas *et al.* abuse timing side-channel leakage from `DIV` instruction used to perform modular reduction—this vulnerability is also absent in SEAL [27]. Pessl *et al.* improve the attack of Primas *et al.* [25]. This attack may target constant-time NTT implementations as in SEAL but scale inefficiently for large polynomials used in FHE. Our proposed attack is simpler and more efficient compared to this attack because it specifically targets ternary value assignments.

The proposed attacks in this work are also different from the earlier *single-trace* analysis of FHE [3] because the earlier attack focuses on Gaussian sampling operations that are replaced in SEAL v3.6. By contrast, our target is another operation and it is shown on the latest version of SEAL to date (v4.1). Moreover, our *single-trace* attack is fundamentally different from multi-trace attacks, which can target FHE's decryption operations. We do not address such attacks on decryption in this study since they are relatively straightforward extensions of the recent multi-trace analysis of lattice-based cryptography [29, 31]. *Single-trace* attacks are known to break defenses such as masking that are built for such multi-trace attacks [27].

An earlier version of this article has been published in the proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security [1]. The major contributions of the conference edition were:

- We reveal a new *single-trace* side-channel leakage of SEAL. We show the processing in NTT function leaks information about the ternary values that can lead to recovering the secret keys in FHE. This vulnerability exists in the latest version (v4.1) of SEAL as of date.
- To effectively extract the side-channel information from a *single-trace* measurement, we propose a two-stage neural network based side-channel profiler. We use two distinct ML classifiers and ensemble results by multiplying guessing scores to improve the guessing success.
- We perform the proposed attack on the ARM Cortex-M4F running SEAL software. The results show that our proposed attack extracts each secret key coefficient with 98.3% accuracy.
- We evaluate random delay insertion countermeasure. We show that random delay insertion defense is susceptible to attacks.

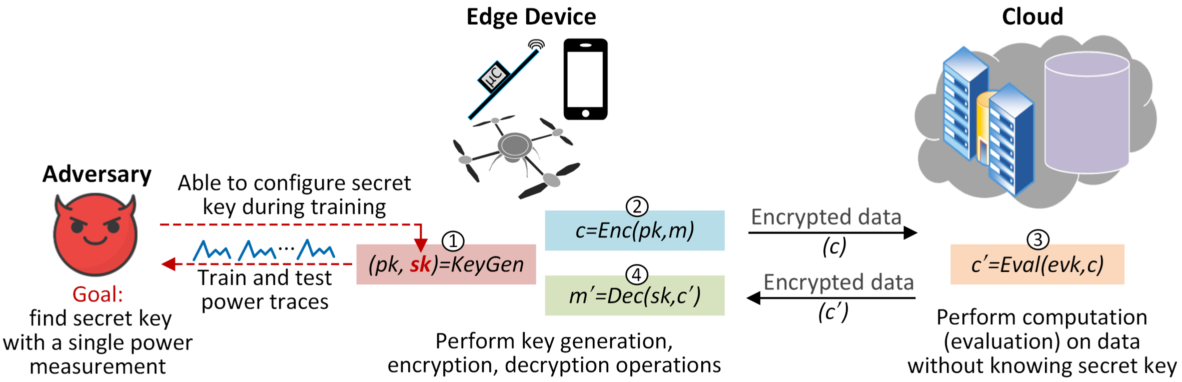
Edge Device
µC
Cloud
Adversary
Able to configure secret key during training
Train and test power traces
Goal: find secret key with a single power measurement
① (pk, sk)=KeyGen
② c=Enc(pk,m)
④ m′=Dec(sk,c′)
Encrypted data (c)
Encrypted data (c′)
③ c′=Eval(evk,c)
Perform key generation, encryption, decryption operations
Perform computation (evaluation) on data without knowing secret key

Fig. 1: Homomorphic encryption functions at the edge device and the cloud. The public key (*pk*) encrypts the message (*m*) to generate the ciphertext (*c*) and the secret key (*sk*) decrypts the received homomorphically evaluated ciphertext (c') – both operations execute on the edge device while homomorphic evaluations execute in the cloud. Our attack executes on the edge devices with obtained physical measurements when the keys are getting generated.

This journal article enhances our prior work in several aspects as outlined below:

- We demonstrate a novel side-channel leakage of SEAL when compiler optimization `-O3` flag is enabled. The `guard` and `mul_root` operations (at lines 3 and 10 in Fig. 8) read data from memory via load (`ldrd`) instructions, which can leak secret information in NTT function. As of date, this vulnerability exists in the latest version (v4.1) of SEAL.
- We evaluate the proposed attack on the ARM Cortex-M4F running SEAL software. We compile the implementation using `gcc-arm-none-eabi` with optimization flag `-O3`. Based on the results, our proposed attack is more effective in extracting each secret key coefficient than our previous attack. We can reveal each secret key coefficient with 98.6% accuracy.

The remainder of the paper is organized as follows. Section 2 provides background information about FHE, NTT, and our threat model. Section 3 then introduces the proposed ML-based side-channel attack with compiler optimization level `-O0`. Section 4 presents the proposed ML-based side-channel attack with compiler optimization level `-O3`. Section 5 evaluates attack results and countermeasures. Subsequently, Section 6 discusses drawbacks of our attack. Finally, Section 7 concludes the work.

2 Preliminaries

This section provides background information about FHE, NTT, ML-based side-channel attacks and threat model.

2.1 Fully Homomorphic Encryption (FHE)

FHE schemes are characterized by four primary functions: key generation, encryption, evaluation, and decryption. Fig.1 illustrates an example structure of HE. The key generation generates secret and public keys.

The encryption uses the public key to encrypt user's message. The evaluation takes the encrypted message and uses the public evaluation key to perform homomorphic operations over the encrypted message. The encrypted messages can be processed by others who do not know the secret key. Decryption takes the secret key and evaluation output to recover the resulting plaintext value.

There are various software and hardware implementations of FHE such as SEAL [30], SEAL-Embedded [24], HElib [16], HEAAN [7], and PALISADE [26]. We specifically focus on SEAL which is compatible with SEAL-Embedded—the first FHE library targeted for embedded devices. While SEAL can support

BFV [13] and CKKS [7] schemes of FHE, SEAL-Embedded only supports the CKKS scheme. In this work, our target scheme is CKKS since it is supported by both SEAL and SEAL-Embedded libraries.

CKKS scheme of FHE is constructed based on the Ring Learning with Errors (RLWE) problem [7]. An RLWE sample $b = as{+}e$ is built by sampling a from R_q (which is the residue ring of R modulo q), noise e sampling over R, and secret key s is chosen from a key distribution over R. In SEAL's CKKS scheme, R_3 is used as secret key distribution. In other words, the secret key is produced from a ternary distribution sampling over $\{-1, 0, 1\}^n$ where modulus n is to be the power of two. This generated secret key is then converted to the NTT domain before performing decryption operations.

SEAL and SEAL-Embedded have several parameter settings [24, 30]. In this paper, we have targeted 128-bit security level and $n = 4096$ which is the default setting of SEAL-Embedded.

2.2 Number Theoretic Transform (NTT)

NTT is basically a form of Fast Fourier Transform (FFT) over finite field. It is used to improve the performance of polynomial multiplication. Its representation is denoted as $\hat{x} = \mathrm{NTT}(x) \in \mathbb{Z}_q^n$ where $x = (x_{0},...,x_{n\text{-}1}) \in R_q$ denotes vectors of polynomials over R_q. Its formulation is $\hat{x}_i = \sum_{j=0}^{n-1} x\ \omega^{ij}$ where ω is fixed n's primitive root of unity. The powers of ω are called twiddle factors. In our target library configuration, modulus degree n is 4096. SEAL has 4 prime modulus using n = 4096 for key generation. Prime modulus (q) is 109 (30 + 30 + 30 + 19) bits and its coefficient values are 0x3ED00001, 0x3ED30001, 0x3ED60001, and 0x66001.

Although secret key coefficients can be equal to -1, 0, or 1, these values are converted to {0, 1, q-1} form before NTT operations. SEAL uses primes at most 30-bits; therefore, NTT inputs can be equal to 0, 1, or 0x3ED00000.

Fig.2 illustrates the first few stages of NTT. NTT consists of $\log_2 n$ stages. In each stage, there are butterfly operations that consist of modular multiplication, addition, and subtraction. SEAL uses the Harvey butterfly structure instead of Cooley-Tukey (CT) [9] and Gentleman-Sande (GS) [14]. NTT has `x` and `y` input coefficients. The `x` coefficients go to `guard` function in SEAL which performs a reduction before `addition` and `subtraction` operations of each butterfly. The `y` coefficients go to `mul_root` function which multiplies `y` coefficients with twiddle factors before `addition` and `subtraction` operations of each butterfly.

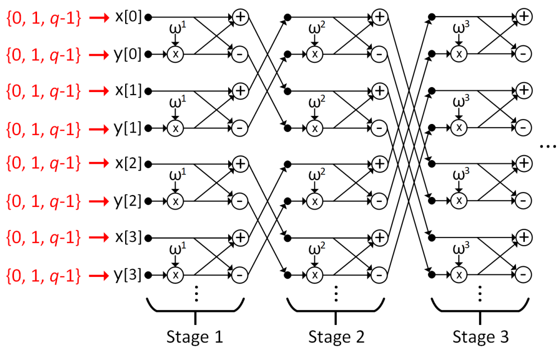

Fig. 2: The first few stages of the NTT. Each stage of NTT consists of multiple butterfly operations. Twiddle factors ω are constant in each stage. NTT inputs of SEAL CKKS scheme can be 0, 1, or q-1.

2.3 Threat Model

This work presents an attack on the NTT operation of SEAL CKKS scheme's key generation to extract secret key coefficients which are invoked by NTT. Fig.1 illustrates that there are edge devices that compute encryption and decryption and a cloud server that computes homomorphic evaluation. Our threat model assumes the adversary has access to the edge device. Therefore, the adversary can capture multiple power traces for building a "profile" of the leakage. We also assume that the adversary knows the executed SEAL code's version and its parameters. Therefore, the adversary can build ML models offline by configuring the device with different keys. During the attack, however, the adversary tries to extract the secret key using *only a single-trace* that is captured from the victim's device. Since the key generation will occur only once for each session, the adversary is limited to a *single* power measurement.

```
1  void transform_to_rev
2  (ValueType *values, int log_n,
3  const RootType *roots,
4  const ScalarType *scalar = nullptr) const{
5    size_t n = size_t(1) << log_n;
6    RootType r;
7    ValueType u, v;
8    ValueType *x = nullptr;
9    ValueType *y = nullptr;
10   std::size_t gap = n >> 1;
11   std::size_t m = 1;
12   ...
13   for (std::size_t i = 0; i < m; i++){
14     r = *++roots;
15     x = values + offset;
16     y = x + gap;
17     for (std::size_t j = 0; j < gap; j+=4){
18       u = arithmetic_.guard(*x);
19       v = arithmetic_.mul_root(*y, r);
20       *x++ = arithmetic_.add(u,v);
21       *y++ = arithmetic_.sub(u,v);
22       ...
23     }
24     offset += gap << 1;
25   }
26   ...
27 }
```

Fig. 3: SEAL's NTT implementation. The highlighted code lines show the lines we target.

3 Proposed Attack with Compiler Optimization Level -O0

This subsection presents the proposed attack and related challenges for compiler optimization disabled settings. We discuss target operations and demonstrate vulnerabilities within the implementation of the target operations.

3.1 Target Operations and Vulnerabilities

Our proposed attack focuses on the NTT which takes SEAL's secret key as input and converts them to the NTT domain during the key generation of FHE. Fig.3 shows the related code scripts of SEAL's NTT implementation. `x` and `y` pointers correspond to secret key coefficients and `r` value corresponds to the twiddle factors. The inner loop performs the butterfly operations of NTT. In each iteration of the inner loop, 4 butterfly operations are executed. The `gap` value is initially equal to 2048 for SEAL's NTT with n = 4096. Therefore, there are 2048 butterfly operations in each stage of NTT.

The first arithmetic operation of NTT is modular reduction operation—`guard` which is shown in line 18 of Fig.3. `x` input coefficients first go through the `guard` function in line 18 of Fig.3. It contains a simple conditional statement that checks whether `x` input is greater than two times modulus ($2q$) or not. If the `x` coefficient is greater than $2q$, it performs a reduction. However, NTT inputs are always in $\{0, 1, q - 1\} < 2q$ in the first stage of NTT. Therefore, this `guard` operation does not change the input values. After the `guard` operation, `y` coefficient and twiddle factor (`r`) go through `mul_root` function in line 19 of Fig.3. The twiddle factors are public values and pre-calculated before the NTT operations. Also, they are smaller numbers in the first few stages of the NTT. Since the twiddle factor is updated outside of the inner loop, it is constant in the inner loop. After the multiplication of the twiddle factor and `y` coefficient, the outputs (`u` and `v`) of `guard` and `mul_root` operations go through `addition` and `subtraction` operations in line 20 and 21 of the Fig.3, respectively.

Our proposed attack targets `addition` and `subtraction` operations which are highlighted in red color in the Fig.3. Since NTT's input coefficients can be 0, 1, or q-1, there are only 9 possible input pairs (*i.e.*, cases). For both `addition` and `subtraction` operations, their inputs (`u` and `v`) in lines 18 and 19 of the Fig.3 depend on NTT's inputs (`x` and `y`). Hence, there are 9 distinct inputs for both `addition` and `subtraction` operations.

3.2 Determining Point of Interest (POI) Regions

A major challenge in performing our proposed attack is finding the points of interest (POI) region of `addition` and `subtraction` operations of each butterfly of NTT. To identify POI regions, we use ML and pre-processing techniques.

Our attack first divides traces into small sampling windows. Each window contains a fixed portion of trace samples and they are labeled as 0 or 1 depending on whether it includes sample points corresponding to the power samples of `addition` and `subtraction` operations or not. Power samples in each window and their corresponding labels are fed to the ML for the training. During the test, power samples in each window are fed to our ML classifier in their natural

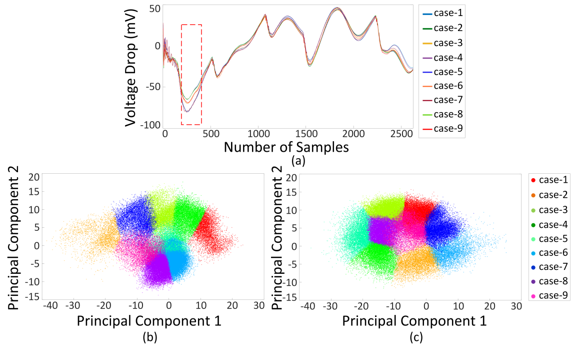

Fig. 4: (a) An example of averaged power trace corresponds to an `addition` operation in the butterfly operation, (b) principal component analysis (PCA) scores for power traces with the samples from 200 to 350 correspond to `addition` operations, (c) PCA scores for power traces with samples from 200 to 350 corresponds to subtraction operations.

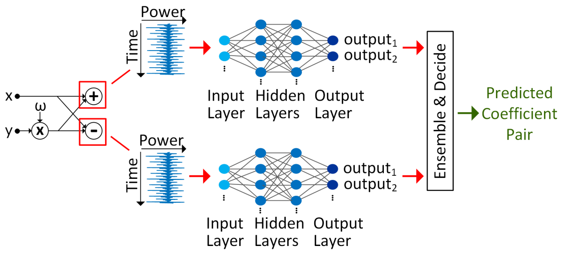

Fig. 5: ML pipeline. Two distinct ML classifiers take the power measurements corresponding to the `addition` and `subtraction` operations in the NTT. The estimated results are then ensembled to predict the NTT's secret key coefficients.

Table 1: An Example of Guessing Scores

case	score for `addition`	score for `subtraction`	ensembled result
1	$1.3448e^{-09}$	$1.1463e^{-06}$	$1.5415e^{-15}$
2	0.5236	0.1470	0.0769
3	$4.1054e^{-04}$	$8.3409e^{-04}$	$3.4242e^{-07}$
4	$5.1324e^{-09}$	$4.3093e^{-07}$	$2.2117e^{-15}$
5	0.4755	0.8339	0.3965
6	$2.9400e^{-05}$	0.0066	$1.9546e^{-07}$
7	$5.7768e^{-08}$	$3.6251e^{-06}$	$2.0942e^{-13}$
8	$4.9694e^{-04}$	0.0109	$5.4059e^{-06}$
9	$7.1847e^{-06}$	$7.7821e^{-04}$	$5.5912e^{-09}$

Correct pair: 5

sequence to identify POI regions corresponding to sequential arithmetic operations of NTT. Then, Pearson correlation coefficient [5] is used to validate POI regions.

The number of power samples in each window affects ML results. When the window size is smaller than the power samples corresponding to the target arithmetic operations, at least one correct guess for the guess of target trace samples can be determined. Therefore, we select the window size as 1000 for both `addition` and `subtraction` operations.

3.3 Exploiting Side-Channel Leakages

The power consumption of the processed data depends on the inputs and operations. To perform a side-channel attack, the adversary needs to model the power consumption of the device. The most well-known power models are Hamming weight, Hamming distance, and identity. According to the used model type, labels of power consumption data can be different. Since there are only 9 possible input pairs of the NTT, we used the identity model and labeled data from 1 to 9.

Fig.4-(a) shows an example of averaged power traces for all 9 input cases of the `addition` operation. The red dashed rectangle—power samples from 200 to 350 in the Fig.4-(a) indicates the highest leakage points—in other words, the power consumption difference for different cases in this region is highest. We use PCA [18] to see the variation of 9 different input pairs. Fig.4-(b) and (c) show the principal component analysis (PCA) scores for power traces with samples from 200 to 350 corresponding to `addition` and `subtraction` operations, respectively. Different colors indicate that data for each pair are grouped in a specific region which means it is not impossible to identify all input pairs statistically.

Our ML-based attack takes the whole power consumption trace corresponding to the target `addition` and `subtraction` operations rather than a specific portion of trace, automatically analyzes all samples of traces, and distinguishes the power traces for all 9 cases.

3.4 Ensembled ML-based Side-Channel Attack

Our proposed attack uses two distinct ML classifiers to estimate input pairs of NTT separately for the `addition` and `subtraction` operations. Fig.5 shows our ML pipeline. Each ML classifier takes power traces corresponding to `addition` and subtraction operations and generates guessing scores. Table 1 shows an example of guessing scores. There are 9 possible guess scores for both addition and subtraction operations. The sum of the scores is 1 for both addition and subtraction operations. The correct pair is 5 in this example. The highlighted line shows that ML classifiers for addition operations guess case-2 with 0.5236 accuracy and ML classifiers for subtraction operations guess case-5 with 0.8339 accuracy. To decide which guessing is correct, our proposed attack ensembles the results by multiplying both guessing scores in each row in Table 1. The highest guessing score in the ensembled result column shows the correct guess which is case-5 in Table 1. Section 5.2 will further provide the hyperparameters of our attack and the attack results.

4 Proposed Attack with Compiler Optimization Level -03

This section presents why previous side-channel leakages do not exist when compiler optimization with -03 flag is enabled. We demonstrate a new side-channel leakage that exists even when compiler optimization is enabled. We also present our ML-based side-channel attack.

4.1 Why Previous Side-Channel Leakages Do Not Exist

Compiler optimization (-03 flag) improves the performance of the code. It performs several optimizations such as common subexpression elimination, loop invariant motion, constant folding, tailcall optimization and tail recursion, conditional execution or branch elimination, function inlining, loop restructuring, instruction scheduling, etc[1]. Fig.6 shows the assembly code for addition and subtraction functions in SEAL's NTT when compiler optimization is disabled. Lines 10-13 and 28-31 of Figure 5 show that ldr and ldrd instructions load inputs from memory into registers for addition and subtraction. ldr and ldrd instructions load inputs of the addition and subtraction functions from memory to registers. However, when compiler optimization is enabled, function inlining happens. Also, since the inputs of addition and subtraction functions are already calculated during guard and mul_root operations and result values are stored to registers, ldr and ldrd instructions do not exist. In this regard, compiler optimization eliminates side-channel leaks caused by loading data from memory to registers.

```
 1  ...
 2  {
 3  _Z14arithmetic_addRyS_:
 4    push   {r4, r5, r7}
 5    sub    sp, #12
 6    add    r7, sp, #0
 7    str    r0, [r7, #4]
 8    str    r1, [r7, #0]
 9      return a + b;
10    ldr    r3, [r7, #4]
11    ldrd   r0, r1, [r3]
12    ldr    r3, [r7, #0]
13    ldrd   r2, r3, [r3]
14    adds   r4, r0, r2
15    adc.w  r5, r1, r3
16    mov    r2, r4
17    mov    r3, r5
18  }
19  ...
20  {
21  _Z14arithmetic_subRyS_:
22    stmdb   sp!, {r4, r5, r7, r8, r9}
23    sub     sp, #12
24    add     r7, sp, #0
25    str     r0, [r7, #4]
26    str     r1, [r7, #0]
27      return a+two_times_modulus_-b;
28    ldr     r1, [r7, #4]
29    ldrd    r4, r5, [r1]
30    ldr     r1, [pc, #44]
31    ldrd    r0, r1, [r1]
32    adds    r2, r4, r0
33    adc.w   r3, r5, r1
34    ldr     r1, [r7, #0]
35    ldrd    r0, r1, [r1]
36    subs.w  r8, r2, r0
37    sbc.w   r9, r3, r1
38    mov     r2, r8
39    mov     r3, r9
40  }
41  ...
```

Fig. 6: Assembly code for addition and subtraction operations in SEAL's NTT implementation when compiler optimization is disabled.

4.2 Target Operations and Vulnerabilities

When compiler optimization is enabled, vulnerabilities of addition and subtraction functions in line 20-21 of Fig.3 does not exist. Therefore,

[1] https://developer.arm.com/documentation/102654/0100/Overview-of-optimizations

```
#define SEAL_COND_SELECT(cond,if_true,
 if_false) (cond ? if_true:if_false)
...
uint64_t guard(const uint64_t &a) const {
 return SEAL_COND_SELECT(a>=two_times_modulus_,
  a-two_times_modulus_, a);
}
...
uint64_t mul_root(const uint64_t &a,
 const MultiplyUIntModOperand &r) const {
 return multiply_uint_mod_lazy(a, r, modulus_);
}
...
uint64_t multiply_uint_mod_lazy(uint64_t x,
 MultiplyUIntModOperand y,
 const Modulus &modulus) {
 unsigned long long tmp1;
 const uint64_t p = modulus.value();
 multiply_uint64_hw64(x, y.quotient, &tmp1);
 return y.operand * x - tmp1 * p;
}
...
```

Fig. 7: The `guard` and `mul_root` functions in SEAL's NTT implementation. Highlighted code shows secret key coefficients that go to functions as parameters.

```
1  ...
2    return SEAL_COND_SELECT(...)
3  ldrd    r5, r6, [r3, #-32]
4  cmp     r5, r8
5  sbcs.w r1, r6, r9
6  bcc.n   0x8000372 ;jump to line 10
7  subs.w r5, r5, r8
8  sbc.w   r6, r6, r9
9   return multiply_uint_mod_lazy(...)
10 ldrd    r4, r1, [r2, #-32]
11 ...
```

Fig. 8: Assembly code for `guard` and `mul_root` operations in SEAL's NTT implementation when compiler optimization (-O3) is enabled. Highlighted code shows the instruction that causes side-channel leakage.

we analyzed the whole code and found new vulnerabilities in NTT's implementation. New vulnerabilities happen due to `guard` and `mul_root` functions in lines 18-19 of Fig.3. The `guard` and `mul_root` functions take secret key coefficients of SEAL's NTT as inputs. The highlighted lines of Fig.7 and the corresponding compiler-generated assembly code with -O3 of Fig.8 show target inputs and instructions that cause side-channel leakage. The `ldrd` instructions read data from memory and load data to registers. Due to the fact that data in memory are secret coefficients of SEAL's NTT, the execution of these `ldrd` instructions results in side-channel leakage.

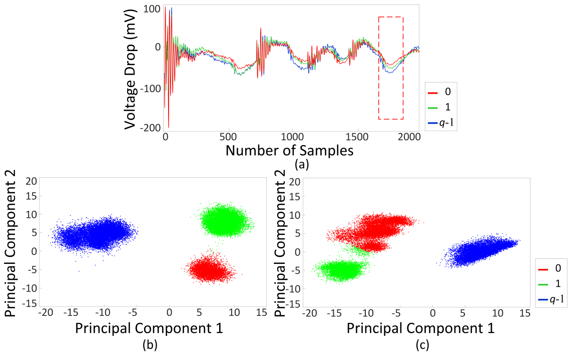

Fig. 9: (a) An example of averaged power trace corresponds to a `mul_root` operation, (b) PCA scores for power traces with the samples that correspond to `ldrd` instruction in guard operations, (c) PCA scores for power traces with samples that correspond to `ldrd` instruction in `mul_root` operations.

Before performing side-channel attacks, we need to find the target POI regions of the power traces as in our earlier attack. Therefore, we follow the same method explained in Section 3.2. Then, we perform PCA to observe variation of data for each input. Fig.9-(a) shows an example of averaged power traces for all 3 secret inputs of the `mul_root` operation. The red-dashed rectangle indicates power samples that correspond to target `ldrd` instruction. Fig.9-(b) and (c) show PCA scores for data that correspond to `guard` and `mul_root` operations, respectively. There is not an insurmountable problem in identifying all secret input coefficients statistically due to the different colors that indicate that the data for each coefficient are grouped in a specific area.

4.3 ML-based Side-Channel Attacks

We perform two different attacks to extract `x` and `y` input coefficients of SEAL's NTT. Our ML classifiers take power traces corresponding to `guard` and `mul_root` operations, respectively. Since each input coefficient can be 0, 1, or q - 1, there are only 3 possible guesses for each classifier. Since ML classifiers guess directly `x` and `y` secret input coefficients in SEAL's NTT instead of input pairs,

Table 2: Network Model Structure and Parameters for `Addition` and `Subtraction` Operations in NTT's Butterfly

	Model for addition		Model for subtraction	
Layer Type	Output Shape	Params. #	Output Shape	Params. #
Input	(None, 2625, 1)	0	(None, 3230, 1)	0
Conv.1D-1	(None, 1314, 64)	320	(None, 1614, 64)	320
MaxPool.1D-1	(None, 657, 64)	0	(None, 807, 64)	0
Conv.1D-2	(None, 654, 128)	32896	(None, 804, 128)	32896
MaxPool.1D-2	(None, 327, 128)	0	(None, 402, 128)	0
Conv.1D-3	(None, 162, 128)	65664	(None, 399, 128)	65664
MaxPool.1D-3	(None, 162, 128)	0	(None, 199, 128)	0
BatchNorm.	(None, 162, 128)	512	(None, 199, 128)	512
Flatten	(None, 20736)	0	(None, 25472)	0
Dropout	(None, 20736)	0	(None, 25472)	0
Dense	(None, 512)	10617344	(None, 512)	13042176
Output	(None, 9)	4617	(None, 9)	4617

Total parameters for addition: 10,721,353
Total parameters for subtraction: 13,146,185

there is no need for an ensembled ML-based side-channel attack as in the previous case. We have two different ML models for guard and `mul_root` operations, respectively. Section 5.3 will further provide the hyperparameters of our attack and the attack results.

5 Experimental Results

This section describes the measurement setup for our experiments and evaluates the proposed attacks with and without compiler enabled settings and a well-known countermeasure for NTT.

5.1 Evaluation Setup

Our evaluation setup uses a development board which contains a 32-bit ARM Cortex-M4F STM32F417IG microcontroller operating at 12 MHz. Due to our proposed attack focusing on the NTT, we only compile the SEAL's NTT code rather than the SEAL's entire code. We compile the code using `gcc-arm-none-eabi` compiler with `-O0` and `-O3` flags. The total memory requirement of the implemented NTT codes is around 75KB RAM and 315KB flash data storage. Since our device supports up to 196KB RAM and 1024KB flash memory, we do not use any external storage. We collect power measurements with a LeCroy WaveRunner 8104 model oscilloscope (with a 1 GS/s sampling rate) using a Riscure current probe [2].

Our ML setup is a workstation with 64 GB of random access memory (RAM), an NVIDIA 1080Ti graphics card, and an Intel i7 9700K processor. We use tensorFlow-gpu 2.8.0 as the backend, with a keras-gpu 2.8.0 front end to train and evaluate ML models.

5.2 Evaluation Results with Compiler Optimization Level `-O0`

To evaluate our proposed ML-based side-channel attack, we collect a total of 90000 power traces. We use 63000, 13500, and 13500 power traces for training, validation, and testing, respectively.

As ML model, we used a convolutional neural network (CNN) architecture which is similar to the work in [19, 21]. Table 2 shows the details of network structures and parameters. There are 3 convolutional and 3 max-pooling layers in total, sequentially as a max-pooling layer after each convolutional layer. After third convolutional and max-pooling layer, there is a batch normalization layer to prevent overfitting on the training. Also, there is a dropout layer that drops connections between neurons with a probability of 0.5 following the batch normalization. The model uses a flatten layer to convert data into a fully layer. There are 2 fully connected layers, including the output layer which has 9 neurons. Output layer uses *Softmax* activation function [6] whereas the remaining layers use *RELU* activation functions [23].

In our proposed attack, feeding the power samples to ML classifiers in the correct order is crucial. If ML classifiers are fed with random train and test data sets, the guessing scores of ML classifiers can correspond to different input pairs of NTT. To solve this issue, we first randomize power traces corresponding to both `addition` and `subtraction` operations at the same time. Then, we split traces for training and testing. Finally, ML classifiers are fed the power traces sequentially. In this way, each guessing score for both `addition` and `subtraction` operations matches with their corresponding input pair.

[2] https://www.riscure.com/product/current-probe

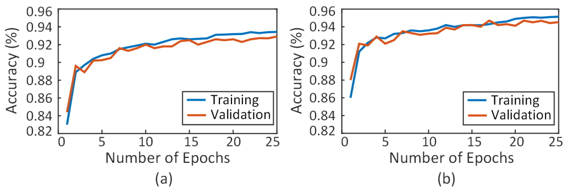

Fig. 10: (a) Training and (b) validation accuracy vs number of epochs for ML models of `addition` and `subtraction` operations, respectively.

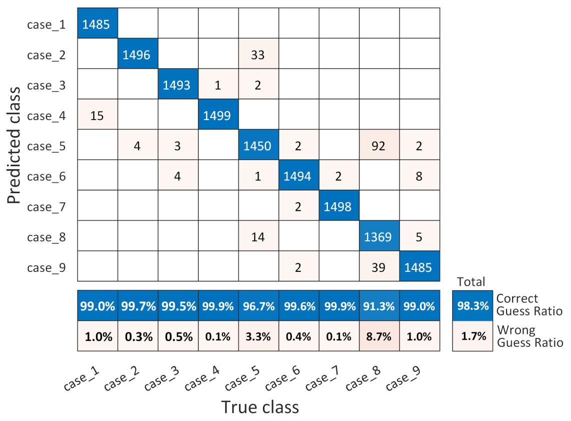

Fig. 11: Confusion matrix of ensembled ML-based side-channel attack with 25 epochs.

Fig.10(a) and (b) show the results of the classification of the models trained with the power traces corresponding to `addition` and `subtraction` operations. When the number of epochs increases, training accuracy converges slowly and reaches around 93% and 95% for ML models of `addition` and `subtraction` operations, respectively.

Fig.11 shows the confusion matrix of our proposed ensembled ML-based side-channel attack with 25 epochs and 63000, 13500, and 13500 power traces, respectively, for training, validation, and testing. When we individually perform ML-based side-channel attacks for `addition` and `subtraction` operations, the correct guess ratios are 92% and 94%, respectively. With our ensembled ML-based side-channel attack, the total correct guess ratio increases to 98.3%.

5.3 Evaluation Results with Compiler Optimization Level `-O3`

Since we have only three possible guesses (three labels), we use a total of 30000 power traces to evaluate our proposed ML-based side-channel attack on `guard` and `mul_root` operations, respectively. We use 21000, 4500, and 4500 power traces for training, validation, and testing, respectively.

Our ML models are based on the CNN architecture, as presented in Section 5.2. The network structure is similar to the network structure that is shown in Table 2. The main difference is that the output layer contains only three nodes in our new attack network because ML classifiers predict directly secret input coefficients of NTT which can be 0, 1, or q - 1.

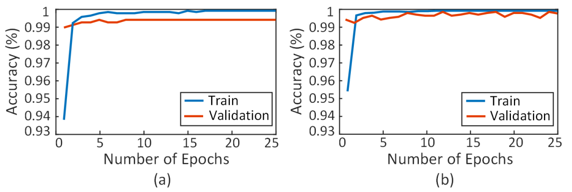

Fig. 12: (a) Training and (b) validation accuracy vs number of epochs for ML models of `guard` and `mul_root` operations, respectively.

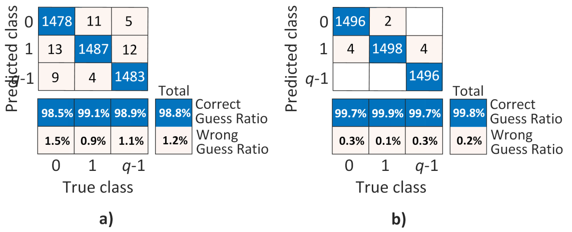

Fig. 13: Confusion matrix of ML-based side-channel attack with 25 epochs for a) `guard` and b) `mul_root` operations.

Fig.12(a) and (b) show the results of the classification of the models trained with the power traces corresponding to `addition` and `subtraction` operations. When the number of epochs increases, training accuracy converges and reaches 100% for both ML models of `guard` and `mul_root` operations.

Fig.13(a) and (b) show the confusion matrix of our ML-based side-channel attack with 25 epochs for `guard` and `mul_root` operations, respectively. When we individually perform ML-based side-channel attacks for `guard` and `mul_root` operations, the correct guess ratios are 98.8% and 99.8%, respectively. Since we target directly `x` and `y` secret input coefficients in SEAL's NTT, there is no need for an ensembled ML-based side-channel attack.

```
for(std::size_t j = 0; j < gap; j+=4) {
    delay_function();
    u = arithmetic_guard(*x);
    delay_function();
    v = arithmetic.mul_root(*y, r);
    delay_function();
    *x++ = arithmetic_.add(u, v);
    delay_function();
    *y++ = arithmetic_.sub(u, v);
    ...
}
```

Fig. 14: Random delay insertion between arithmetic operations in NTT.

We consider the worst-case scenario to state our attack success rate for NTT's each secret coefficient. To calculate the worst-case guess rate, we sum up the wrong guess rates (1.2% and 0.2%), which equals 1.4%. As a result, the total attack success rate for NTT's each secret coefficient is 98.6%.

5.4 Random Delay Insertion Countermeasure

Random delay insertion method which generates random delays in embedded software increases the attacker's uncertainty about the location of the target operation [10, 11]. To implement this countermeasure into NTT, we write a delay function that selects a random number between 0 and pre-selected threshold value and generates a delay depending on the selected random number. During the delay duration, `NOP` executes in the processor. We add the delay function between each arithmetic operation of NTT shown in Fig.14.

To evaluate the random delay insertion countermeasure, we find the position of target operations with ML and then perform the side-channel attack to extract the NTT's secret key. Our attack first divides the power traces into equal trace windows, and then labels the power traces in binary format like in Section 3.2. For example, power trace windows corresponding to the `addition` operation are labeled as 1 and the remaining trace windows are labeled as 0. ML takes power traces and labels to build ML models. Since ML can estimate wrong results and false positives, the selection of window size by dividing power traces is very crucial. We select the size of window as 1000 that is smaller than the size of power traces corresponding to the target operations. Since the sample size is 2625 for `addition` and 3230 for `subtraction` operation, there are 3-4 and 4-5 sequential windows labeled as 1 for each `addition` and `subtraction` operation, respectively. ML estimates at least one correct guess for each target point. Then, target POI regions are identified using these ML guess results. Therefore, this countermeasure is not resistant to side-channel attacks.

6 Discussions

In this work, we set the operating frequency of the device to 12 MHz. If we increase the operating frequency, the noise of the platform will increase. Hence, attacking may require a great number of traces to build ML models. There are multiple prior works [2, 17] which demonstrate *single-trace* side-channel attacks with lower frequencies, including 8 MHZ to attack on NTT [25].

SEAL supports different configurations with different parameters. Our attack focuses on its 128-bit security level with $n = 4096$. However, depending on the selected configuration setting, there will be a different number of NTT operations and prime modulus. Therefore, we have to build new ML models to perform the attack.

Since the goal of our work is to expose side-channel vulnerabilities of the SEAL and perform a *single-trace* attack on it, we did not concentrate on implementing a resistant countermeasure to our attack. Shuffling countermeasures can be considered a secure defense mechanism to protect the NTT [28]. We intend to implement it in the future.

7 Conclusions

In this work, we propose new *single-trace* side-channel attacks on an FHE library, SEAL, with real power measurements. Our first proposed attack exploits leakage of `addition` and `subtraction` operations of SEAL's NTT with compiler optimization `-O0` level—non-optimized code implementation used, e.g., for constant-time enforcement in cryptography. Specifically, we demonstrate a vulnerability in the NTT a side-channel leakage coming from the NTT's `addition` and `subtraction` operations and perform an ensembled ML-based side-channel attack on it. We show that we are able to extract SEAL's secret key coefficients with ensembled ML-based side-channel attack with 98.3% accuracy. The second attack targets `guard` and `mul_root` operations with compiler optimization `-O3` level. We show that the side-channel leakage coming from the NTT's `addition` and `subtraction` operations do not exist when compiler `-O3` optimization is enabled. We demonstrate our second attack can extract SEAL's secret key coefficients with a 98.6% accuracy by targeting `guard` and `mul_root` operations in SEAL's NTT. Furthermore, we evaluate random delay insertion countermeasure and show that the random delay insertion countermeasure is not a suitable countermeasure to protect the NTT against our ML-based attacks.

8 Ethical Disclosures

We contacted the Cryptography and Privacy Research Group at Microsoft Research to report our preliminary findings and disclosed this paper before publication.

9 Acknowledgments

This research is based upon work supported by the National Science Foundation under the Grants No. CNS 16-2137283 – Center for Advanced Electronics through Machine Learning (CAEML) and its industry members.

References

[1] Aydin F, Aysu A (2022) Exposing side-channel leakage of seal homomorphic encryption library. In: Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security (ASHES), pp 95–100

[2] Aydin F, Aysu A, Tiwari M, et al (2021) Horizontal side-channel vulnerabilities of post-quantum key exchange and encapsulation protocols. ACM Transactions on Embedded Computing Systems 20(6):1–22. https://doi.org/https://doi.org/10.1145/3476799

[3] Aydin F, Karabulut E, Potluri S, et al (2022) RevEAL: Single-trace side-channel leakage of the SEAL homomorphic encryption library. In: 2022 Design, Automation and Test in Europe Conference & Exhibition (DATE), pp 99–117, https://doi.org/10.23919/DATE54114.2022.9774724

[4] Boemer F, Lao Y, Cammarota R, et al (2019) nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, pp 3–13

[5] Brier E, Clavier C, Olivier F (2004) Correlation power analysis with a leakage model. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp 16–29

[6] Campbell D, Dunne R, Campbell NA (1997) On The Pairing Of The Softmax Activation And Cross–Entropy Penalty Functions And The Derivation Of The Softmax Activation Function. In: Australian Conference on Neural Networks, pp 181–185

[7] Cheon J, Kim A, Kim M, et al (2017) Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT), pp 409–437

[8] Colombier B, Grosso V, Cayrel PL, et al (2023) Horizontal Correlation Attack on Classic McEliece. IACR Cryptol. ePrint Arch., Report 2023/546

[9] Cooley J, Tukey JW (1965) An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation 19(90)

[10] Coron JS, Kizhvatov I (2009) An efficient method for random delay generation in embedded software. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp 156–170

[11] Coron JS, Kizhvatov I (2010) Analysis and improvement of the random delay countermeasure of CHES 2009. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp 95–109

[12] Drucker N, Pelleg T (2022) Timing leakage analysis of non-constant-time NTT implementations with Harvey butterflies. In: International Symposium on Cyber Security, Cryptology, and Machine Learning (CSCML), pp 99–117

[13] Fan J, Vercauteren F (2012) Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive, Report 2012/144

[14] Gentleman W, Sande G, Rohatgi P (1966) Fast fourier transforms: for fun and profit. In: In Fall Joint Computer Conference (AFIPS), pp 563–578

[15] Gentry C (2009) Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, p 169–178

[16] Halevi S, Shoup S (2014) Algorithms in HElib. In: Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, pp 554–571

[17] Huang WL, Chen JP, Yang BY (2019) Power analysis on NTRU Prime. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2019(1):123–151. URL https://doi.org/10.13154/tches.v2020.i1.123-151

[18] Jolliffe IT (2002) Principal Component Analysis, Springer New York, NY, pp 1–488

[19] Kashyap P, Aydin F, Potluri S, et al (2020) 2Deep: Enhancing side-channel attacks on lattice-based key-exchange. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 40(6):1217–1229. https://doi.org/10.1109/TCAD.2020.3038701

[20] Kim I, Lee T, Han J, et al (2020) Novel single-trace ML profiling attacks on NIST 3 round candidate Dilithium. IACR Cryptol. ePrint Arch., Report 2020/1383

[21] Kim J, Picek S, Henuser A, et al (2019) Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2019(3):148–178. URL https://doi.org/10.13154/tches.v2019.i3.148-179

[22] Li Q, Huang Z, Lu W, et al (2020) HomoPAI: A secure collaborative machine learning platform based on homomorphic encryption. In: 2020 IEEE 36th International Conference on Data Engineering, pp 1713–1713

[23] Nair V, Hinton G (2010) Rectified linear units improve restricted Boltzmann machines. In: International Conference on Machine Learning (ICML), pp 807–814

[24] Natarajan D, Dai W (2021) SEAL-embedded: A homomorphic encryption library for the internet of things. IACR Transactions on Cryptographic Hardware and Embedded Systems 2021(3):756–779

[25] Pessl P, Primas R (2019) More practical single-trace attacks on the number theoretic transform. In: International Conference on Cryptology and Information Security in Latin America (LATINCRYPT), pp 130–149

[26] Polyakov Y, Rohloff K, Ryan GW, et al (2022) PALASIDE lattice crypto library. https://gitlab.com/palisade/palisade-release/blob/master/doc/palisade_manual.pdf

[27] Primas R, Pessl P, Mangard S (2017) Single-trace side-channel attacks on masked lattice-based encryption. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp 513–533

[28] Ravi P, Poussier R, Bhasin S, et al (2020) On configurable SCA countermeasures against single trace attacks for the NTT. pp 123–146

[29] Ravi P, Roy S, Chattopadhyay A, et al (2020) Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2020(3):307–335. URL https://doi.org/10.13154/tches.v2020.i3.307-335

[30] SEAL (2022) Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA.

[31] X. Zheng WWA. Wang (2013) First-order collision attack on protected NTRU cryptosystem. Microprocessors & Microsystems 37(6-7):601–609