Timing leakage analysis of non-constant-time NTT implementations with Harvey butterflies

Nir Drucker and Tomer Pelleg

IBM Research - Haifa

Abstract. Harvey butterflies and their variants are core primitives in many optimized number-theoretic transform (NTT) implementations, such as those used by the HElib and SEAL homomorphic encryption libraries. However, these butterflies are not constant-time algorithms and may leak secret data when incorrectly implemented. Luckily for SEAL and HElib, the compilers optimize the code to run in constant-time.
We claim that relying on the compiler is risky and demonstrate how a simple code modification can cause leakage, which can reduce the hardness of the ring learning with errors (R-LWE) instances used by these libraries, for example, from 2^{128} to 2^{104}.

Keywords: NTT, Harvey's Butterflies, Constant-time code, Compiler Optimizations, Ring-LWE, Side-Channel Attacks

1 Introduction

Constant-time implementations are today considered a requirement for cryptographic libraries that provide production-level code. Commonly, an implementation is considered "constant-time" if code branches and memory access patterns are independent of secret information. Efficiently achieving this property is not always easy, and the literature is full of examples that exploit optimized code that does not run in constant-time, for example, [12, 14, 22].

The number-theoretic transform (NTT) algorithm is used by many cryptographic implementations to achieve fast polynomial multiplications. Some examples include post-quantum schemes such as Kyber [24], NTTRU [21], and Dilithium [11], or homomorphic encryption (HE) libraries such as SEAL [18], HELib [15], Palisade [27], and HEAAN [7]. Despite the performance benefits provided by NTT, it is still the bottleneck in many of these implementations, which makes it an ideal target for optimization. The list of works that deal with NTT optimizations in hardware and software is large. In this work, we analyze the constant-time property in the optimizations of [16, 19] that are also used by SEAL, HElib, and other software [3, 16, 17] and hardware [10, 23] implementations. Specifically, we consider Harvey butterflies [16] that involve branches for performing fast modular reduction and therefore are not considered constant-time algorithms. Nevertheless, some constant-time implementations for them exist, such as the implementation of [19] [Section 5] and the vectorized implementations of [3].

SEAL uses the following C macro to perform the modular reduction branch of Harvey's butterfly:

```c
#define SEAL_COND_SELECT(cond,if_true,if_false) (cond ? if_true:if_false)
```

and states that "This is a temporary solution that generates constant-time code with all compilers on all platforms." We tested this claim on Linux for all the supported compilers "Clang++ (≥ 5.0) or GNU G++ (≥ 6.0)" and observed that the Assembly code generated by the compiler indeed always used a conditional move (CMOV) instruction.

The same ternary pattern used in SEAL_COND_SELECT is used in NTL [26], an optimized mathematical library that HElib uses for its NTT implementation. However, unlike SEAL, NTL also includes a branch-less implementation that is controlled by the flag NTL_AVOID_BRANCHING. The reason for including the flag seems to be related to performance and not to security. This is indicated in the code comment:

> "On some modern machines, this is usually faster and NTL uses this non-branching strategy. However, on other machines (modern x86's are an example of this), conditional move instructions can be used in place of branching, and this code can be faster than the non-branching code. **NTL's performance-tuning script will figure out the best way to do this.**"

or in the comment

> "With this option, branches are replaced at several key points with equivalent code using shifts and masks. It may speed things up on machines with deep pipelines and high branch penalties."

As a final example, we consider the Palisade code [27], which does not seem to implement Harvey's butterflies but still uses a simple if-else code.

Despite requests (e.g., [5]) to include a built-in directive-API in GCC that will force compilers to use a conditional move, it does not yet exist. Consequently, SEAL's assumption could be wrong in new compilers and OSs. Without continuous integration tests to test this assumption, secret information can be leaked. In fact, this may already happen today. The SEAL function multiply_plain_normal performs multiplication of plaintext by HE ciphertext, with cases where the plaintext is a secret as indicated therein: "Optimizations for constant/monomial multiplication can lead to the presence of a timing side-channel in use-cases where the plaintext data should also be kept private". This function uses the macro SEAL_COND_SELECT, but its translation to Assembly involves branches, which contradicts the original comment.

A side-channel analysis of the NTT algorithm in the context of ring learning with errors (R-LWE) was presented, for example, by [22]. Specifically, this attack relies on a Hamming-weight leakage model, where the data for the model was collected from real traces using electromagnetic (EM) measurements. In contrast, our attack targets high-end CPUs (e.g., x86-64), where collecting EM data is rarely possible. Instead, we rely only on timing differences and specifically the binary knowledge of whether a branch was taken or not. This knowledge can be collected for example when the code is called from within Intel®SGX®by using the SGX-step framework [25]. Another difference between our work and [22] is that our attack focuses on NTT implementations with Harvey butterflies, as in the case for SEAL and HElib code. The work of [22] assumed the following modular reduction operation $a \pmod q = a - q\left\lfloor \frac{a}{q} \right\rfloor$, which allowed them to collect leakage information from the variable-time `DIV` instruction on Cortex-M4F. This knowledge allowed them to report a full key recovery attack on NTT. In contrast, we are restricted to a smaller leakage and therefore report only partial key extraction. Still, this partial key extraction can lead to a reduction in the hardness of the R-LWE instances and should be taken into account by security researchers who evaluate potential risks for using a certain implementation.

Our contribution. This work identifies places in the code of SEAL, HElib and other HE libraries that depend on the compiler for generating a constant-time code. We show why this assumption is risky and analyze the security loss caused by the potential leakage in the key generation code of SEAL, which uses NTT with Harvey butterflies. Our analysis shows that if the generated Assembly is not a constant-time code, we can extract more than 9% of the secret key, which reduces the hardness of the R-LWE instance by more than 10%. For example, from 2^{128} to 2^{104} security estimation.

Organization. The document is organized as follows. Section 2 provides some background and describes our notation. We describe the risk of depending on the compiler in Section 3. Section 4 analyzes the leaked data in the case of a sparse secret and reports our results. We conclude in Section 6.

2 Background and notation

Let $\mathbb{F}_q$ be a finite field of characteristic q with residue classes represented as elements from $Z \cap [0, q)$. The elements in the polynomial quotient ring $\mathcal{R}_q = \mathbb{F}_q[X]/(X^N+1)$ are polynomials of a degree at most $N-1$ with integer coefficients in $\mathbb{F}_q$, where $q \equiv 1 \pmod{2N}$ and N is a power of two. We may refer to a polynomial $a = \sum a_i x^i$ by its coefficients i.e., $a = (a_0, \ldots, a_{N-1})$. For a specific platform, we denote its word-size with β. For example, for typical CPUs $\beta = 2^{32}$ or $\beta = 2^{64}$. We use $\wedge$ to denote logical-and.

2.1 Distribution of an HE secret key

Modern HE schemes rely on the R-LWE assumption [20]. The plaintext, key, and error domains are the polynomials ring $\mathcal{R}_q$, where the keys and errors are randomly derived from the χ_{key}, χ_{err} distributions, respectively. Let $R_q^u \subset \mathcal{R}_q$ be the set of all polynomials from $\mathcal{R}_q$ with coefficients in $\{0, \pm 1\}^N$, then the commonly used options for χ_{key} are the

1. uniform distribution over R_q^u
2. uniform distribution over R_q^u with Hamming weight h, for a positive integer h
3. distribution over R_q^u, where each coefficient has a probability $\frac{\alpha}{2}, \frac{\alpha}{2}, 1-\alpha$ of being $+1, -1, 0$, respectively

The χ_{err} distribution is commonly a Gaussian distribution. We target the residue number system (RNS) variant of CKKS [6] and its key generation method, where a secret key s is sampled from χ_{key}. The original CKKS variant [6] sets χ_{key} according to option #2 with $h = 64$. In contrast, SEAL implements option #1 and HElib implements option #3 with $\alpha = 0.5$. In SEAL, the function `generate_sk` calls the function `sample_poly_ternary` and in HElib, the function `SecKey::GenSecKey` calls `sampleSmallBounded`. We demonstrate the potential leakage on SEAL and set χ_{key} according to option #1, which simplifies the attack computations. We conjecture that similar exploits can be generated for the other distributions. After generating the secret key s, the SEAL `generate_sk` function invokes the NTT algorithm on s. This behavior is common to other libraries as well.

2.2 NTT

The NTT algorithm is a variant of the fast Fourier transform (FFT) algorithm over $\mathcal{R}_q$. It receives a polynomial $a = (a_0, \ldots, a_{N-1}) \in \mathcal{R}_q$ and a fixed N's primitive root of unity ω as inputs; it outputs $\tilde{a} = (\tilde{a}_0, \ldots, \tilde{a}_{N-1}) \in \mathcal{R}_q$, where $\tilde{a}_i = \sum_{j=0}^{N-1} a_j \omega^{ij}$. The inverse function $a = InvNTT_\omega(\tilde{a})$ is given by $a_i = \frac{1}{N}\sum_{j=0}^{N-1} \tilde{a}_j \omega^{-ij}$. The powers of ω are called twiddles.

Appendix A presents one variant of the NTT and inverse-NTT (InvNTT) algorithms as specified in [19]. These algorithms are implemented in different libraries such as SEAL. Variants of these algorithms are available in NTL and inherently in HElib. The main bottleneck of these algorithms is the Cooley-Tukey (CT) [8] and Gentleman-Sande (GS) [13] butterflies, respectively. These are implemented in SEAL using Harvey butterflies [16], which we present in Algorithm 1. For brevity, we use $\texttt{ShoupModMul}(t, \omega, \omega', q) = \omega t - q \left\lfloor \frac{\omega' t}{\beta} \right\rfloor$ to denote Shoup's multiplication [16], which performs a lazy reduction and leaves the output in the range $[0, 2q-1]$. Here, β is a fixed global parameter, and $\omega' = \left\lfloor \frac{\omega\beta}{q} \right\rfloor$ is a precomputed value.

3 Compiler optimizations

The compiler's decision to use a conditional move or branch depends on the penalty that the compiler believes a program hits when it executes the specific

Algorithm 1 Harvey's Butterflies [16]

```
    Global parameters: A word-size $\beta$, a modulus $q < \frac{\beta}{4}$; $\omega \in \mathbb{F}_q$; $\omega' = \left\lfloor \frac{\omega\beta}{q} \right\rfloor < \beta$
    Input: $0 \leq x, y < 4q$
    Output: $x = x + \omega y,\ y = x - \omega y \pmod{4q}$
1: procedure HarveyFwdButterfly($x, y, \omega, \omega', q, \beta$)
2:     if $x \geq 2q$ then $x = x - 2q$
3:     $t =$ShoupModMul$(y, \omega, \omega', q)$
4:     return $(x + t,\ x - t + 2q)$
    Input: $0 \leq x, y < 2q$
    Output: $x = x + y,\ y = \omega(x - y) \pmod{2q}$
1: procedure HarveyInvButterfly($x, y, \omega, \omega', q, \beta$)
2:     $x' = x + y$
3:     if $x' \geq 2q$ then $x' = x' - 2q$
4:     $t = x - y + 2q$
5:     $y' =$ShoupModMul$(t, \omega, \omega', q)$
6:     return $x', y'$
```

branch. Controlling this penalty and observing the changes in the compiler output is possible using the compiler's target flag `-mbranch_cost=`x, where x is in $\{0, \ldots, 5\}$. Indeed, when setting $x \leq 2$ and compiling the SEAL code, the output Assembly does not include conditional moves.

The reasons that led the compiler to add conditional moves to begin with, are performance-related and not security-related. Thus, other flags may affect its decision. For example, the following GCC optimization flags `-O0`, `-fno-if-conversion`, `-fno-if-conversion2`, `-fno-tree-loop-if-convert`, and `-fno-tree-loop-if-convert-stores` may turn off this optimization.

Someone may inadvertently compile a cryptographic library using the above flags, for example in debug mode. It can also be the case that an adversary intentionally injects these compilation flags in order to convert the code be non constant-time. Putting these two options aside, it is still interesting to explore some examples where the compiler simply does not know how to compute the branch penalty and thus even when using the default optimization mode, it uses branches because this is its default behavior. Consider the next example:

```
uint64_t foo(uint32_t *a) {
    uint64_t i = 0;
    while (i < 5) {
        i = (i >= a[i]) ? i+2 : i;
    }

    return i;
}
```

Here, the compiler does not know the content and size of a and thus the generated Assembly (using Clang-10) is

```
 1    0:    31 c0                   xor    %eax,%eax
 2    2:    eb 12                   jmp    16 <foo+0x16>
 3    4:    66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 4    b:    00 00 00
 5    e:    66 90                   xchg   %ax,%ax
 6   10:    48 83 f8 05             cmp    $0x5,%rax
 7   14:    73 0e                   jae    24 <foo+0x24>
 8   16:    8b 0c 87                mov    (%rdi,%rax,4),%ecx
 9   19:    48 39 c8                cmp    %rcx,%rax
10   1c:    72 f2                   jb     10 <foo+0x10>
11   1e:    48 83 c0 02             add    $0x2,%rax
12   22:    eb ec                   jmp    10 <foo+0x10>
13   24:    c3                      retq
```

which involves the jb branch on line 10, instead of a conditional move. Another, perhaps simpler example, is the function

```
uint64_t foo2(uint32_t a, uint32_t b) {
    return (a > b ? a : b);
}
```

which when compiled with GCC-9 uses conditional moves (cmovb).

```
  44:    39 fe                   cmp    %edi,%esi
  46:    0f 42 f7                cmovb  %edi,%esi
  49:    89 f0                   mov    %esi,%eax
  4b:    c3                      retq
```

However, once we perform a simple modification to it

```
uint64_t foo3(uint32_t a, uint32_t b) {
    if (b < 100000) return b;
    return (a > b ? a : (a < 2*b ? b : a));
}
```

the output assembler involves branches.

```
  54:    39 f7                   cmp    %esi,%edi
  56:    77 10                   ja     68 <foo3+0x18>
  58:    8d 04 36                lea    (%rsi,%rsi,1),%eax
  5b:    39 f8                   cmp    %edi,%eax
  5d:    76 09                   jbe    68 <foo3+0x18>
  5f:    89 f0                   mov    %esi,%eax
  61:    c3                      retq
  62:    66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
  68:    89 f8                   mov    %edi,%eax
  6a:    c3                      retq
```

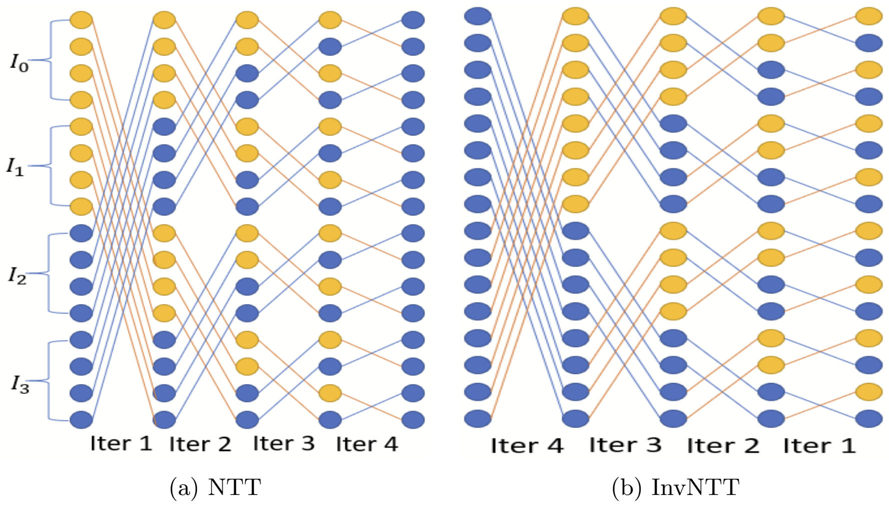
Note that the logic in foo3 is the same logic as in foo2. The false condition of the first ternary operator should be considered only when $a \leq b$; in which case it also follows that $a \leq 2b$. In this example, for the logic to stay the same, it is important that the $2 \cdot b$ operation does not result in an integer overflow. To accommodate this, we added the first if statement to bound b. Another example involves lookup tables where the key is a secret, as in [9].

The above examples show that even a small code modification may break SEAL's assumption as already happened for the function multiply_plain_normal. Therefore, even if a code is currently compiled with a conditional move, it is important to understand the consequences of a compilation mistake that leads to leaking secret information.

4 Exploiting NTT over secret keys

We already saw how a simple code modification or a simple malicious injection of a compilation flag can result in a variable time implementation that may leak secret information. In this section, we provide an example that exploits a faulty NTT implementation. Specifically, we target the NTT transformation that SEAL applies to every secret key. We focus on this scenario for two reasons. First, the distribution of the key is over polynomials with small coefficients, hereafter denoted sparse polynomials. Second, this example uses the forward NTT of [19], which involves a smaller number of twiddles in its first few iterations. This is in contrast to analyzing the inverse NTT of [19], which is more complex since it involves a different twiddle for every butterfly.

Although we focus on the SEAL code, the methods we apply here should work with minor modifications on other key distributions, butterflies, or NTT implementations. For example, we could have performed the same analysis to extract data on the encryption error, which is derived from a somewhat wider distribution.



(a) NTT (b) InvNTT

Fig. 1: The first few iterations of the NTT (panel a) and invNTT (panel b) algorithms over polynomials of degree $N = 16$. Yellow circles (x) demonstrate values that will go through the branch $x > 2q\ ?\ x\ :\ x - 2q$ in a subsequent iteration.

In this demonstration, we analyze, measure, and accumulate the leakage knowledge after every NTT iteration. We use analytical methods for the first and second iterations and empirical methods for the other iterations, in which the number of options to analyze grows exponentially. Figure 3 shows the first few iterations of the NTT and InvNTT algorithms over polynomials of degree $N = 16$. The yellow circles (x) demonstrate values that go through the branch $x > 2q \ ? \ x \ : \ x - 2q$ from which we attempt to extract information.

Observation 1 *For a sparse polynomial input, no information is leaked from the branches of the first NTT iteration.*

Proof. The NTT inputs are always in $\{0, 1, q-1\} < 2q$. □

Second iteration. For the second iteration, we first define the variable $s =$ `ShoupModMul`$(\mathbf{q-1}, \omega, \omega', q)$ and observe some of its properties in Lemmas 1, 2. Note that $s \neq 0, q$ by definition.

Lemma 1. *For a given set of parameters β, ω, q of the NTT algorithm (Alg. 3). When*

$$\omega\beta \pmod q < \frac{(q-\omega)}{(q-1)}\beta$$

$s < q$ otherwise $q < s \leq 2q$.

Proof. First, we compute

$$\begin{aligned}\frac{(q-1)\omega'}{\beta} &= \frac{(q-1)\left\lfloor\frac{\omega\beta}{q}\right\rfloor}{\beta} = \frac{q\left\lfloor\frac{\omega\beta}{q}\right\rfloor}{\beta} - \frac{\left\lfloor\frac{\omega\beta}{q}\right\rfloor}{\beta} \\ &= \frac{\omega\beta - (\omega\beta \pmod q)}{\beta} - \frac{\omega\beta - (\omega\beta \pmod q)}{\beta q} \\ &= \omega - \left[\frac{\omega\beta + (q-1)(\omega\beta \pmod q)}{\beta q}\right] = \omega - \eta\end{aligned}$$

and observe that $\left\lfloor\frac{(q-1)\omega'}{\beta}\right\rfloor = \omega - 1$ when $0 \leq \eta \leq 1$ as in our assumption,

$$\omega\beta \pmod q < \frac{(\beta q - \beta\omega)}{(q-1)} = \frac{(q-\omega)}{(q-1)}\beta$$

We obtain the first claim by

$$\begin{aligned}s &= \texttt{ShoupModMul}(q-1, \omega, \omega', q) = \omega(q-1) - q\left\lfloor\frac{\omega'(q-1)}{\beta}\right\rfloor \\ &= \omega(q-1) - q(\omega-1) = q - \omega < q\end{aligned}$$

When $1 < \eta < 2$ we have $\left\lfloor\frac{(q-1)\omega'}{\beta}\right\rfloor = \omega - 2$ and $q \leq s = 2q - w < 2q$. Note that $\eta < 2$; otherwise $s > 2q$, which contradicts Harvey's proof [16][Theorem 1]. □

Lemma 2. *For a positive integer γ, when $q < \frac{\beta}{\gamma}$ and $\omega < \frac{\gamma-1}{\gamma}q + \frac{1}{\gamma}$, it follows that $s < q$.*

Proof. Lemma 1 states that $s < q$ when

$$\omega\beta \pmod{q} < \frac{(q-\omega)}{(q-1)}\beta \tag{1}$$

but $\omega\beta \pmod{q} < q$, which implies that $s < q$ at least for $q < \frac{(q-\omega)}{(q-1)}\beta$. Because $q < \frac{\beta}{\gamma}$, the previous equations holds when

$$\frac{\beta}{\gamma} < \frac{(q-\omega)}{(q-1)}\beta$$

which after rearrangement becomes

$$\omega < \frac{(\gamma-1)}{\gamma}q + \frac{1}{\gamma}$$

□

Example 1. For HarveyFwdButterfly, $q < \frac{\beta}{4}$. Thus, $\omega < 0.75q + 0.25 \Rightarrow s < q$.

Example 2. For $\beta = 2^{64}$ and $q < 2^{32}$, it holds that $\omega < \frac{2^{32}-1}{2^{32}}q + \frac{1}{2^{32}} \Rightarrow s < q$.

The last example emphasizes that in SEAL, where the implementation uses 64-bit scalars ($\beta = 2^{64}$), for primes that are orders of magnitude smaller than 2^{64} we will rarely encounter the $q < s < 2q$ case.

Algorithm 2 Second iteration exploit

```
    Input: $s$ (see text), branches parameters $br_k$, $br_{k+2^{m-1}}$.
    Output: Possible input pairs for the coefficients at position $k$ and $k + 2^{m-1}$
 1: procedure SecondIterExploit($s$, $br_k$, $br_{k+2^{m-1}}$):
 2:     $out = \{0, 1, q-1\} \times \{0, 1, q-1\}$
 3:     if $br_k$ is taken: then
 4:         return $\{(q-1, q-1)\}$
 5:     if $br_{k+2^{m-1}}$ is taken then
 6:         $out = \{(0,0), (1,0), (q-1,0), (q-1,1)\}$
 7:         if $s < q$ then
 8:             $out = out \cup \{(q-1, q-1)\}$
 9:     if $br_{k+2^{m-1}}$ is not-taken then
10:         $out = \{(0,1), (0,q-1), (1,1), (1,q-1)\}$
11:         if $s > q$ and $br_k$ is unknown then
12:             $out = out \cup \{(q-1, q-1)\}$
13:     return $out$
```

Theorem 1. *Given the NTT parameters $q, \omega, \omega', \beta$ and $N = 2^m$, and $k \leq 2^{m-1}$, for the input variable $s =$* `ShoupModMul`$(q-1, \omega, \omega', q)$ *and the branches br_k and $br_{k+2^{m-1}}$, Algorithm 2 returns a list of the possible coefficients at position k and $k + 2^{m-1}$ after applying* `HarveyFwdButterfly` *on a sparse polynomial.*

Proof. Let $x_k, x_{k+2^{m-1}}$ be the inputs to HarveyFwdButterfly and denote by $x'_k, x'_{k+2^{m-1}}$ its output. Table 1 shows the possible outputs of the `HarveyFwdButterfly` after the first NTT iteration.

Table 1: Possible x'_k (left) and $x'_{k+2^{m-1}}$ (right) values.

	x_k		
$x_{k+2^{m-1}}$	0	1	$q-1$
0	0	1	$q-1$
1	ω	$\omega+1$	$\omega+q-1$
$q-1$	s	$s+1$	$s+q-1$

	x_k		
$x_{k+2^{m-1}}$	0	1	$q-1$
0	$2q$	$2q+1$	$2q+q-1$
1	$2q-\omega$	$2q-\omega+1$	$2q-\omega+q-1$
$q-1$	$2q-s$	$2q-s+1$	$2q-s+q-1$

First, recall that $\omega \notin \{1, q-1\}$ as there are no primitive roots of unity for primes bigger than 3 and that $0 < \omega, s < 2q$. Then, by looking at the table, we see that the only case for b_k to be taken ($x'_k > 2q$, Step 3) is when $(x_k, x_{k+2^{m-1}}) = (q-1, q-1)$ and $q < s < 2q$. We continue the analysis assuming that br_k is not taken or is unknown. Here,

$$\begin{aligned}
(x_k, x_{k+2^{m-1}}) \in \{(0,0),(1,0),(q-1,0),(q-1,1)\} &\Rightarrow (x'_{k+2^{m-1}} \geq 2q) \\
&\Rightarrow b_{k+2^{m-1}} \texttt{ is taken} \\
(x_k, x_{k+2^{m-1}}) \in \{(0,1),(0,-1),(1,1),(1,-1)\} &\Rightarrow (x'_{k+2^{m-1}} < 2q) \\
&\Rightarrow b_{k+2^{m-1}} \texttt{ is not taken} \\
((x_k, x_{k+2^{m-1}}) = (q-1, q-1)) \wedge (s < q)& \\
&\Rightarrow b_{k+2^{m-1}} \texttt{ is taken} \\
((x_k, x_{k+2^{m-1}}) = (q-1, q-1)) \wedge (q < s < 2q)& \\
&\Rightarrow (b_{k+2^{m-1}} \texttt{ is not taken}) \wedge (b_k \texttt{ is unknown})
\end{aligned}$$

The correctness of the algorithm follows. □

4.1 Extracted leakage after the second iteration

Algorithm 2 provides us with a way to reduce the number of options for the inputs x_k and $x_{k+2^{m-1}}$ of the NTT algorithm. Theorem 2 summarizes the expected leakage and Figure 2 illustrates it. The theorem uses the following four intervals $I_0 = [0, \frac{N}{4})$, $I_1 = [\frac{N}{4}, \frac{N}{2})$, $I_2 = [\frac{N}{2}, \frac{3N}{4})$, $I_3 = [\frac{3N}{4}, N)$.

Theorem 2. *Let the input to Algorithm 3 be a sparse polynomial and let $0 \leq \rho \leq 1$ be the percentage of the extracted branches (distributed uniformly).*

1. *The probability that a coefficient (x_k, $k \in I_0 \cup I_2$) has only one option when $q < s < 2q$ is $P_1 = \frac{\rho}{9}$.*
2. *The probability for a coefficient (x_k) to have only two options is*

$$P_2 = \begin{cases} \frac{4\rho}{9} & (s < q) \wedge (k \in I_0 \cup I_2) \\ \frac{4\rho^2}{9} & (q \leq s < 2q) \wedge (k \in I_0) \\ \frac{3\rho^2+5\rho}{9} & (q \leq s < 2q) \wedge (k \in I_2) \end{cases}$$

Proof. We define the following events:

$$E_1 := (br_{k+2^{m-1}} \texttt{ is not taken }) \mid (0 < s < q)$$
$$E_2 := (br_k \texttt{ is taken}) \mid (q < s < 2q)$$
$$E_3 := (br_k \texttt{ is not taken}) \wedge (br_{k+2^{m-1}} \texttt{ is not taken }) \mid (q < s < 2q)$$
$$E_4 := (br_k \texttt{ is not taken}) \wedge (br_{k+2^{m-1}} \texttt{ is taken }) \mid (q < s < 2q)$$
$$E_5 := (br_k \texttt{ is unknown}) \wedge (br_{k+2^{m-1}} \texttt{ is not taken }) \mid (q < s < 2q)$$

where Algorithm 2 outputs

$$\begin{aligned} \{(0,1),(0,q-1),(1,1),(1,q-1)\} &\texttt{ for } E_1, E_3 \\ &\Rightarrow x_k \in \{0,1\} \texttt{ and } x_{k+2^{m-1}} \in \{1, q-1\} \\ \{(0,0),(1,0),(q-1,0),(q-1,1)\} &\texttt{ for } E_4 \\ &\Rightarrow x_{k+2^{m-1}} \in \{1, q-1\} \\ \{(0,1),(0,q-1),(1,1),(1,q-1),(q-1,q-1)\} &\texttt{ for } E_5 \\ &\Rightarrow x_{k+2^{m-1}} \in \{1, q-1\} \\ \{(q-1,q-1)\} &\texttt{ for } E_2 \\ &\Rightarrow x_k = x_{k+2^{m-1}} = q-1 \end{aligned}$$

The coefficients of sparse polynomials are uniformly distributed and independent of ρ. Thus, the above events will happen with probability

$$Pr(E_1) = \frac{4\rho}{9} \quad Pr(E_2) = \frac{\rho}{9} \quad Pr(E_3) = Pr(E_4) = \frac{4\rho^2}{9} \quad Pr(E_5) = \frac{5\rho(1-\rho)}{9}.$$

At the second iteration we only perform branches over inputs at positions $I_0 \cup I_2$ (see illustration in Figure 3), which limits the leakage to the coefficients x_k, $k \in I_0 \cup I_2$ of the original input polynomial to the NTT algorithm. When $q < s < 2q$, the probability that a coefficient (x_k, $k \in I_0 \cup I_2$) has only one option is $P_1 = Pr(E_2) = \frac{\rho}{9}$.

The probability of a coefficient having two options is

$$P_2 = \begin{cases} Pr(E_1) = \frac{4\rho}{9} & (s < q) \wedge (k \in I_0 \cup I_2) \\ Pr(E_3) = \frac{4\rho^2}{9} & (q \leq s < 2q) \wedge (k \in I_0) \\ (Pr(E_3) + Pr(E_4) + Pr(E_5)) = \frac{3\rho^2+5\rho}{9} & (q \leq s < 2q) \wedge (k \in I_2) \end{cases}$$

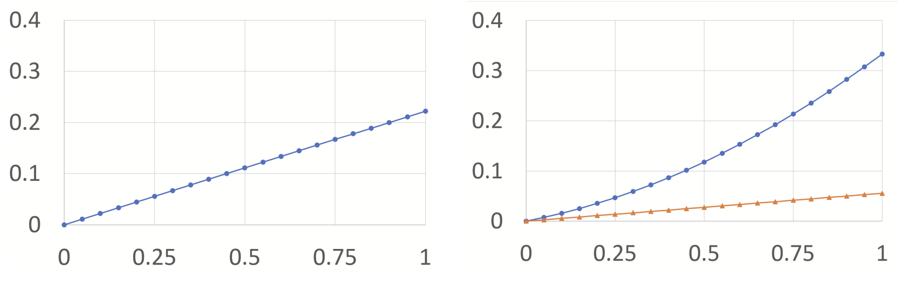
□

Let X_1, X_2 denote the number of coefficients with only one or two options, respectively, for a given ρ. In addition, for a polynomial of degree N and $\rho = 1$, we define $Y_1 = \mathbb{E}(X_1)/N$, $Y_2 = \mathbb{E}(X_2)/N$ to be the portion of coefficients we identified to have a reduced number of options (one or two, respectively).

Corollary 1. *For an input of a sparse polynomial of degree N, and a fixed ρ,*

$$\mathbb{E}(X_1) = \begin{cases} 0 & s < q \\ \frac{\rho}{18}N & q < s < 2q \end{cases}$$

$$\mathbb{E}(X_2) = \begin{cases} \frac{2\rho}{9}N & s < q \\ \left(\frac{1}{4} \cdot \frac{4\rho^2}{9} + \frac{1}{4} \cdot \frac{3\rho^2+5\rho}{9}\right) N = \frac{7\rho^2+5\rho}{36}N & q < s < 2q \end{cases}$$



(a) When $0 < s < q$ (b) When $q < s < 2q$

Fig. 2: Second iteration leakage for a given branch extraction probability ρ. The orange triangles show the portion of coefficients (out of N) that are completely identified (Y_1). The blue dots show the portion of coefficients (out of N) that now have only two options out of three (Y_2).

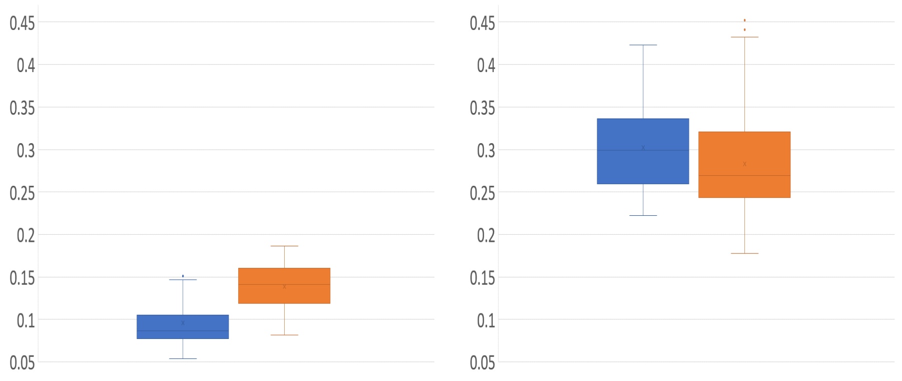
Third iteration. The input to the third iteration is the output of the second iteration, which we can view as the output of a radix-4 NTT, where every branch depends on a coefficients-quartet of the original polynomial x_k, $x_{k+2^{m-2}}$, $x_{k+2\cdot 2^{m-2}}$, $x_{k+3\cdot 2^{m-2}}$ for $k \in I_0$. For every such quartet we may know two branch results from the second iteration. In addition, at the third iteration, for half the quartets $0 \le k < 2^{m-3}$, we add the knowledge of all four branches, while for the other half, we learn nothing.

Using an analytical approach, as we did for analyzing the leakage at the second iteration, is more complex because the number of cases increases exponentially. For example, in the second iteration, we only had two cases where we needed to consider whether or not $s < q$.

To assess the complexity of analyzing the third iteration, we performed an exploratory empirical experiment, which revealed more than 80 cases that need to be analyzed. We note that, unlike post-quantum schemes where a prime is usually fixed by design, many HE implementations generate the required prime numbers on-the-fly and according to the users' needs. For these reasons, we decided to use an empirical approach to demonstrate the potential data leakage from the third iteration. We stress that this can be fine-tuned when the set of primes is predefined (fixed).

Our experiment involves a program, which for a given modulo N and a prime q, computes the minimal $2N$ primitive root of unity ω. It then executes the first three iterations of the NTT, brute-forcing over all possible quartet inputs. The program generates a branching table as we did for the second iteration (Table 2) and outputs the answers to the following questions:

1. What is the probability of this branch combination occurring, assuming uniform distribution of the original coefficients?
2. How many coefficients does it reduce to one or two options (Y_1, Y_2)?

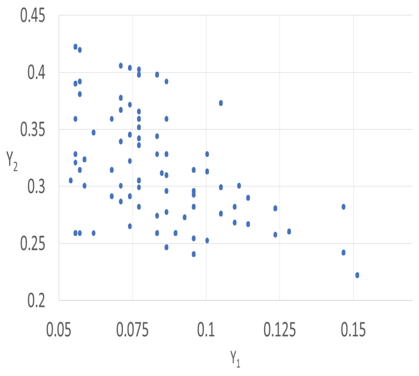


(a) Y_1 (b) Y_2

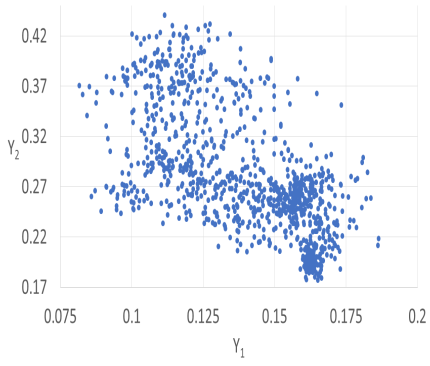
Fig. 3: Third and fourth iterations leakage extraction (Y_1, Y_2). Blue and orange box-plots show the distribution results when using information up to the third and fourth iteration, respectively.

We repeated the above process for the fourth iteration, where every branch affects eight inputs at a time.

For our experiments we generated $1,000$ NTT-friendly primes using the code from Appendix B, and fixed $N = 2^{15}$. Figure 3 shows the portion of coefficients we identified as having a reduced number of options (one or two). As these numbers depend on the values of q and N, we use box-plots to demonstrate the resulting distributions. The left and right box-plots show the distributions after the third and fourth iterations, respectively. As expected, at the fourth iteration, Y_1 increases on average while Y_2 decreases. This phenomenon is also



(a) Third Iteration



(b) Fourth Iteration

Fig. 4: Correlation between Y1 and Y2, using the information from the third and fourth iterations. The number of dots indicates the number of cases to consider for the experimented primes.

demonstrated in Figure 4, where we see a negative correlation between Y_1 and Y_2.

Moving into deeper iterations. It is possible to extend the search to deeper iterations. However, the number of options that our program checks grows super-exponentially. In the ith iteration, it considers 2^{i-1} coefficients per one of the two tables, with three options per coefficient. Thus, the total number of options to consider is $3^{2^{i-1}}$. For example, for $i = 5$ and $i = 6$ the number of options per table to consider is $\sim 2^{25}$ and $\sim 2^{50}$, respectively.

4.2 Hardness of RLWE instances after the leakage

Modern HE libraries follow the HE standard [1] when considering security parameters. The standard, in turn, uses the *LWE estimator* [2] to compute the relevant values. We followed the standard and re-evaluated the security guarantees provided when using the same values of N and $\log_2 q$. Specifically, we used the command

```
n = 2048; q = 2^54; alpha = 8/q; m = 2*n
estimate_lwe(n, alpha, q, secret_distribution=(-1,1),
             reduction_cost_model=BKZ.sieve, m=m)
```

from [2], which is designed for SEAL. To assess the security impact, we only considered the coefficients that we completely extracted by replacing n with $n - nY_1$ in the script above. Table 2 summarizes the results. For the second iteration, we used $Y_1 = \frac{\rho}{18}$ from Theorem 2. For the third and forth iterations, we first evaluated the entropy per experiment using the equation

$$\frac{(Y_2 + \log_2 3 \cdot (1 - Y_1 - Y_2))}{\log_2 3}$$

and then chose the pairs (Y_1, Y_2) that yielded the minimal and maximal entropy per iteration. Specifically, we used $(0.0555556, 0.259259)$ and $(0.146605, 0.2824075)$ for the third iteration, and $(0.08916325, 0.2458845)$ and $(0.17332525, 0.35162325)$ for the fourth iteration. It can be observed that when including the leakage from the fourth iteration, the security estimate drops from 2^{128} to around $2^{104} - 2^{115}$, an 18% gap. Taking Y_2 into consideration will result in an even higher estimation for the drop in security.

Table 2: Estimated security level of HE instances after combining data from our extracted leakage. For our baseline we use N and q for 2^{128} classical bits security from [1].

N	$\log_2 q$	Security estimation		
		2nd iter.	3rd iter.	4th iter.
$1,024$	27	2^{124}	2^{111} - 2^{124}	2^{107} - 2^{119}
$2,048$	54	2^{121}	2^{109} - 2^{121}	2^{105} - 2^{117}
$4,096$	109	2^{120}	2^{107} - 2^{120}	2^{103} - 2^{115}
$8,192$	218	2^{120}	2^{107} - 2^{120}	2^{104} - 2^{115}
$16,384$	438	2^{120}	2^{108} - 2^{120}	2^{104} - 2^{115}
$32,768$	881	2^{120}	2^{108} - 2^{120}	2^{104} - 2^{115}

5 Responsible Disclosure

The paper was disclosed to the HEAAN, HELib, Palisade and SEAL teams before its publication.

6 Conclusions

In this work, we identified a potential vulnerability that can occur in almost all HE libraries, where the NTT algorithm and specifically the Harvey butterflies can be compiled to have a non constant-time Assembly. While there is, currently, no vulnerability when the libraries are used as intended, we explained how a simple code modification or even a simple misuse of the library (e.g., compiling and using it in debug mode) can flip the situation. Specifically, we evaluated the potential leakage of a secret from the NTT implementation of SEAL and showed the hardness degradation of the R-LWE instances used therein.

While it is not always possible to control the compiler results, we recommend that library owners either use hand-written Assembly for critical code paths or modify the code so it won't use branches. Such code exists, for example, in HElib when compiled with the flag `NTL_AVOID_BRANCHING` or in the vectorized implementation of [3,4].

A NTT algorithms

Algorithms 3 and 4 are the forward and inverse NTT algorithms from [19], respectively.

Algorithm 3 CT radix-2 NTT [19]

```
    Input: $a \in \mathcal{R}_q$, $N$ a power of 2, $q$ a prime satisfying $q \equiv 1 \pmod{2N}$, $\psi_{rev}$, which
    holds the powers of $\psi$ in bit-reversed order.
    Output: $\tilde{a} = NTT_{\psi}(a)$ in bit-reversed order.
 1: procedure CT RADIX-2 NTT($a, N, q, \psi_{rev}$)
 2:     $t = N$, $\tilde{a} = a$
 3:     for ($m = 0$; $m < N$; $m = 2m$) do
 4:         $t = t/2$
 5:         for $i = 0$; $i < m$; $i$++ do
 6:             $w = \psi_{rev}[m + i]$
 7:             for ($j = 2it$; $j < (2i + 1)t$; $j$++) do
 8:                 $(X, Y) = (\tilde{a}_j, \tilde{a}_{j+t}w)$
 9:                 $(\tilde{a}_j, \tilde{a}_{j+t}) = (X + Y, X - Y) \pmod{q}$
10:     return $\tilde{a}$
```

Algorithm 4 Gentleman-Sande (GS) Radix-2 InvNTT [19]

```
    Input: $\tilde{a} \in \mathcal{R}_q$, $N$ a power of 2, $q$ a prime satisfying $q \equiv 1 \pmod{2N}$, $\psi^{-1}_{rev}$, which
    holds the powers of $\psi^{-1}$ in bit-reversed order.
    Output: $a = InvNTT(\tilde{a})$ in bit-reversed order.
 1: procedure GENTLEMAN-SANDE (GS) RADIX-2 INVNTT($\tilde{a}, N, q, \psi^{-1}_{rev}$)
 2:     $t = 1$, $a = \tilde{a}$
 3:     for ($m = N/2$; $m > 0$; $m \gg= 1$) do
 4:         for ($i = 0$; $i < m$; $i$++) do
 5:             $w = \psi^{-1}_{rev}[m + i]$
 6:             for $j = 2it$; $j < (2i + 1)t$; $j$++) do
 7:                 $(X, Y) = (a_j, a_{j+t})$
 8:                 $(a_j, a_{j+t}) = (X + Y, w(X - Y)) \pmod{q}$
 9:         $t = 2t$
10:     for ($j = 0$; $j < N$; $j$++) do
11:         $a_j = a_j \cdot N^{-1}$
12:     return a
```

B Generating the primes

For reproduction purposes, we provide the SageMath script we used to generate the primes in Section 4.

```
import numpy as np
import math
import random
# Generate a random prime p in [n+1, n^3+1],
#  where p = 1 mod n
def primeGen(n):
    i=0;
    while True:
        x = randrange(n^2);
        if is_prime( x*n +1):
            return x*n+1
random.seed(120)
primes = [primeGen(2^16) for i in range(1000)]
```

References

1. Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., Vaikuntanathan, V.: Homomorphic encryption security standard. Tech. rep., HomomorphicEncryption.org, Toronto, Canada (November 2018), https://homomorphicencryption.org/standard/
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169–203 (2015). https://doi.org/doi:10.1515/jmc-2015-0016
3. Boemer, F., Kim, S., Seifu, G., de Souza, F.D., Gopal, V.: Intel HEXL : Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. Tech. rep. (2021), https://eprint.iacr.org/2021/420
4. Bradbury, J., Drucker, N., Hillenbrand, M.: NTT software optimization using an extended Harvey butterfly. Tech. rep. (2021), https://eprint.iacr.org/2021/1396
5. gcc bugs: [Bug c++/98801] New: Request for a conditional move built-in function (2021), https://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg676288.html
6. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A Full RNS Variant of Approximate Homomorphic Encryption. In: Cid, C., Jacobson Jr., M.J. (eds.) Selected Areas in Cryptography – SAC 2018. pp. 347–368. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-10970-7_16
7. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15
8. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19**(90), 297–301 (1965). https://doi.org/10.2307/2003354

9. Daan, S.: LLVM provides no side-channel resistance (2019), https://dsprenkels.com/cmov-conversion.html
10. Dai, W., Sunar, B.: cuHE: A Homomorphic Encryption Accelerator Library. In: Pasalic, E., Knudsen, L.R. (eds.) Cryptography and Information Security in the Balkans. pp. 169–186. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-29172-7_11
11. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (2017), https://pq-crystals.org/dilithium/data/dilithium-specification.pdf
12. Espitau, T., Fouque, P.A., Gérard, B., Tibouchi, M.: Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against StrongSwan and Electromagnetic Emanations in Microcontrollers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1857–1874. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134028
13. Gentleman, W.M., Sande, G.: Fast fourier transforms—For fun and profit. AFIPS Conference Proceedings - 1966 Fall Joint Computer Conference, AFIPS 1966 pp. 563–578 (1966). https://doi.org/10.1145/1464291.1464352
14. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on frodokem. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020. pp. 359–386. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_13
15. Halevi, S., Shoup, V.: Algorithms in helib. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I. Lecture Notes in Computer Science, vol. 8616, pp. 554–571. Springer (2014). https://doi.org/10.1007/978-3-662-44371-2_31
16. Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002
17. Jung, W., Lee, E., Kim, S., Lee, K., Kim, N., Min, C., Cheon, J.H., Ahn, J.H.: HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization (2020)
18. Laine, K.: Simple Encrypted Arithmetic Library 2.3.1. Tech. rep., Microsoft, WA, USA (2017), https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf
19. Longa, P., Naehrig, M.: Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. pp. 124–139. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_8
20. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. J. ACM **60**(6) (nov 2013). https://doi.org/10.1145/2535925
21. Lyubashevsky, V., Seiler, G.: NTTRU: Truly Fast NTRU Using NTT **2019**, 180–201 (May 2019). https://doi.org/10.13154/tches.v2019.i3.180-201
22. Primas, R., Pessl, P., Mangard, S.: Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. pp. 513–533. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_25

23. Sadegh Riazi, M., Laine, K., Pelton, B., Dai, W.: HEAX: An architecture for computing on encrypted data. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS pp. 1295–1309 (2020). https://doi.org/10.1145/3373376.3378523
24. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck M., J., Seiler, G., Stehle, D.: CRYSTALS-KYBER (2020), https://pq-crystals.org/kyber/
25. Van Bulck, J., Piessens, F., Strackx, R.: SGX-Step: A practical attack framework for precise enclave execution control. In: 2nd Workshop on System Software for Trusted Execution (SysTEX). pp. 4:1–4:6. ACM (Oct 2017). https://doi.org/10.1145/3152701.3152706
26. Victor, S.: NTL – a library for doing numbery theory – version 11.5.1, commit 91acd5b3a7df709c0d8bf88a99a24bc340dc34f7 (2021), https://github.com/libntl/ntl
27. Yuriy, P., Kurt, R., Gerard, R.W., Dave, C.: PALISADE Lattice Cryptography Library, commmit d76213499af44558170cca6c72c5314755fec23c (2021), https://gitlab.com/palisade/palisade-release