Cache-Timing Attack on the SEAL Homomorphic Encryption Library
Wei Cheng, Jean-Luc Danger, Sylvain Guilley, Fan Huang, Amina Bel Korchi, Olivier Rioul

To cite this version:
Wei Cheng, Jean-Luc Danger, Sylvain Guilley, Fan Huang, Amina Bel Korchi, et al.. Cache-Timing Attack on the SEAL Homomorphic Encryption Library. 11th International Workshop on Security Proofs for Embedded Systems (PROOFS 2022), Sep 2022, Leuven, Belgium. hal-03780506

HAL Id: hal-03780506
https://telecom-paris.hal.science/hal-03780506
Submitted on 19 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Cache-Timing Attack on the SEAL Homomorphic Encryption Library

Wei Cheng$^{1,2}$, Jean-Luc Danger$^1$, Sylvain Guilley$^{2,1}$, Fan Huang$^{3,1}$, Amina Bel Korchi$^2$, and Olivier Rioul$^1$

$^1$ LTCI, Télécom Paris, Institut Polytechnique de Paris, 91120, Palaiseau, France, firstname.lastname@telecom-paris.fr

$^2$ Secure-IC S.A.S., 75014, Paris, France, sylvain.guilley@secure-ic.com

$^3$ University of Science and Technology of China, 230026, Hefei, China, lanplush@mail.ustc.edu.cn

Abstract

Homomorphic encryption (HE) ensures provable secrecy of data processed in the ciphertext domain. However, it happens that FHE private-key algorithms can be broken by side-channel attacks. We disclose a novel cache-timing attack on the SEAL open-source HE library. It is triggered by a non-constant time Barrett modular multiplication, which is one of the building blocks in SEAL. We both analyze the mathematical conditions upon which the leakage occurs and show the experimental feasibility of the attack.

Keywords: Homomorphic encryption; Barrett modular multiplication; Extra-reduction; Side-channel attack; Cache-timing attack; SEAL C++ library.

1 Introduction

No one can deny the massive increase in data over the years, leading to an increased need for storage capacity. Thus, cloud computing becomes one of the most important Information Technology (IT) services for industry to improve business. It allows not only the storage but also the processing of data.

Nevertheless, a significant security issue is raised with the remote manipulation of data inside the Cloud. Hence, a very challenging topic appears, which is to find a solution enabling manipulating data while ensuring their protection, privacy, and anonymization. Homomorphic encryption responds to this challenge and enables computations, without decryption, on encrypted data. Let $E(a)$ and $E(b)$ be the encryption of $a$ and $b$ using an homomorphic cryptosystem, $E(a)$ and $E(b)$ verify the following properties:

- $E(a) \oplus E(b) = E(a + b)$ \hfill // Homomorphy in addition,

- $E(a) \otimes E(b) = E(a \times b)$ \hfill // Homomorphy in multiplication.

There are two variants of homomorphic encryption: Fully Homomorphic Encryption (FHE) and Somewhat Homomorphic Encryption (SWHE). While FHE is a fully homomorphic encryption that allows the evaluation of an arbitrary circuit, SWHE can only evaluate circuits of constant depth. The circuit depth correspond to the number of multiplications that could be performed within a given scheme. Beyond this depth, decryption is not possible because of the noise that is produced during plaintext encryption. This noise increases after each multiplication of the ciphertexts until reaching a level where the decryption is not possible.

In 2009, Gentry [9] has invented the first FHE cryptosystem using a bootstrapping [9] technique to transform a SWHE cryptosystem into a FHE cryptosystem. The security of Gentry’s scheme is based on ideal lattices [13]. This procedure allows transforming a ciphertext
resulting from a circuit to a fresh ciphertext including a noise similar to the one in a new encrypted ciphertext.

Following Gentry’s cryptosystem, many schemes have been proposed as in [7, 10, 25], using different security hardness assumptions. However, FHE schemes are still not practical due to the time to turn on the bootstrapping technique. For this reason, many SWHE schemes as [4, 5, 8] have been developed to allow using homomorphic encryption in practice. Most of these schemes are based on LWE [20] (Learning with errors) and RLWE [12] (Ring Learning with Errors) problems. FHE algorithms are typically being standardized at an international level [23].

In this paper, we choose to study the case of Brakerski-Fan-VercauTen (BFV) homomorphic encryption scheme [5, 8], since it is one of the most known and practical SWHE schemes. SEAL\(^1\) [21] is one of the existing libraries implementing this cryptosystem. It is an open source library, developed in 2015, in C++ and C# by a team from Microsoft, without any dependency on external libraries. In addition, SEAL has been tailed to SEAL-Embedded [16] with small code and memory footprint for resource-constrained embedded devices, e.g., for ARM Cortex M4 platform.

Contributions. We target Barrett modular multiplication implemented in SEAL library and exhibit a timing leakage caused by the extra-reductions. The condition of occurring extra-reductions is refined in SEAL case. We then exploit the timing leakage to launch a key-recovery attack and demonstrate by simulation that the secret key can be extracted using a few ciphertexts with extra-reductions.

Outline. The reminder of this paper is organized as follows. Sec. 2 introduces preliminaries, followed by theoretical timing leakage analysis in Barrett multiplication in Sec. 3. The cache-timing analysis is presented in Sec. 4 and finally Sec. 5 concludes the paper.

2 Preliminaries

2.1 Notation

Let \(\mathbb{Z}_q\) the integers modulo \(q\). The algebraic structure used in BFV scheme is the polynomial ring \(R_q = \mathbb{Z}_q[X]/\phi(X)\), with \(\phi(X) = X^n + 1\). We note \(R = \mathbb{Z}[X]/f(X)\) where \(\mathbb{Z}[X]\) is the polynomial ring with coefficients in \(\mathbb{Z}\) and \(\phi(X)\) is a cyclotomic polynomial. Elements of \(R\) are polynomials of degree less than \(n\) and coefficients in \(\mathbb{Z}\). Elements of \(R_q = \mathbb{Z}_q[X]/\phi(X)\) are polynomials of degree less than \(n\) and coefficients modulo \(q\), where \(\mathbb{Z}_q[X]\) is the polynomial ring with coefficients modulo \(q\). Elements of the ring \(R_q\) are noted in lowercase \((a \in R_q)\), we denote by \([a]_q\) the elements in \(R\) obtained by computing all its coefficients modulo \(q\), \(a + b\) (resp. \(a \cdot b\)) is the addition (resp. multiplication) of two polynomials \(a\) and \(b\) in \(R_q\). In the case where \(a\) and \(b\) are two vectors in \(R_q^l\) with elements \(a_i\) and \(b_i\) in \(R_q\), the addition of \(a\) and \(b\) is \(a + b\) is a vector of \(l\) elements \(a_i + b_i\) in \(R_q\) and the canonical scalar product is used for the multiplication \(a \cdot b\) which is a vector of \(l\) elements \(a_i b_i\) in \(R_q\). For \(x \in \mathbb{R}\), we denote by [\(x\)] rounding to the nearest integer, \([x]\), and [\(x\)] rounding up and down, and then the fractional part of \(x\) is denoted as \{\(x\}\}, that is \{\(x\}\} = x - [x]. At last, let \(\ell\) be the bit-length of a word or a variable, e.g., \(\ell_q = \lfloor \log_2(q) \rfloor + 1\).

\(^1\)Available at https://www.microsoft.com/en-us/research/project/microsoft-seal/.
In the following, we recall the Learning With Errors (LWE) and Ring-LWE (RLWE) problems, and then the BFV scheme.

2.2 LWE/R-LWE Problems


2.3 BFV Scheme

The cryptosystem of BFV is a somewhat homomorphic encryption scheme, developed in 2012, its security is based on the hardness of the RLWE problem [20].

The problem is described in the mathematical ring formed by degree $d$ polynomials over a finite field such as the integers modulo a prime number $q$. Let $\phi(x)$ be a cyclotomic polynomial of degree $d$, and $q \geq 2$ a modulus depending on a security level $\lambda$. For a random $s \in R_q$ and a distribution $\chi = \chi(x)$ over $R$, the problem consists in distinguishing $(a, [a \cdot e + s]_q)$ from a random pair sampled uniformly from $R_q \ast R_q$, where $a$ is a random element of $R_q$ and $e$ a noise term from $\chi$.

2.3.1 Key Generation

As BFV is a public key cryptosystem, the key generation returns a public key $p_k$ and a secret key $s_k$ as described in Alg. 1.

Algorithm 1: Key Generation

| Input: | params = $(R, d, q, t, \chi_{\text{err}}, \chi_{\text{key}})$ |
| Output: | $s_k, p_k, rlk$ |
| 1 | Pick a random $s_k$ uniformly $s_k \leftarrow \chi_{\text{key}}$. |
| 2 | Pick a random $a \leftarrow R_q$. |
| 3 | Sample a random error $e \leftarrow \chi_{\text{err}}$. |
| 4 | Compute $p_k = ([-(a \cdot s_k + e)]_q, a)$. |
| 5 | return $s_k, p_k$ and $rlk$. |

2.3.2 Encryption

The encryption of a message $m \in R_t$ is computed using the public key $p_k$ and returns a ciphertext of two polynomials defined in Alg. 2.

2.3.3 Decryption

The decryption of a ciphertext $C = (c[0], c[1])$ is computed using the secret key $s_k$. The following Alg. 3 describes in details the decryption procedure.

The private key $s_k$ is used in the decryption algorithm. At line 1 it is multiplied by a ciphertext limb $c[1]$. This operation is sensitive. Of course, the result of the (modular) multiplication is not disclosed to the attacker. But, as we shall see in the next section, it reveals
Algorithm 2: Encryption

**Input:** $m \in R_t$ and $p_k$

**Output:** $E(m) = (c[0], c[1])$

1. Compute $\delta = \lfloor \frac{q}{t} \rfloor$
2. Sample $u \leftarrow \chi_{key}$
3. Sample an error $e_1 \leftarrow \chi_{err}$
4. Sample an error $e_2 \leftarrow \chi_{err}$
5. Compute $c[0] = \lfloor p_0 \cdot u \rfloor_q$
6. Compute $r_0 = \lfloor c[0] + e_1 \rfloor_q$
7. Compute $c[0] = \lfloor r_0 + \delta \cdot m \rfloor_q$
8. Compute $r_0 = \lfloor p_1 \cdot u \rfloor_q$
9. Compute $c[1] = \lfloor r_0 + e_2 \rfloor_q$
10. $E(m) = (c[0], c[1])$
11. **return** $E(m)$

Algorithm 3: Decryption

**Input:** $C = (c[0], c[1])$ and $s_k$

**Output:** $D(C)$

1. $r_0 = c[1] \cdot s_k$  
   // Target modular multiplication
2. $D(C) = \lfloor c[0] + r_0 \rfloor_q$
3. $r_0 = t \cdot D(C)$
4. $D(C) = \lfloor \frac{r_0}{t} \rfloor_t$
5. **return** $D(C)$

A side-channel, in that this modular multiplication operates in non-constant time. Therefore, although the decryption operation is benign *per se* (it returns information to be disclosed), the analysis of its side-channel can lead to a complete break of the cryptosystem; indeed, an attacker who possesses $s_k$ can basically decrypt (illegitimately) all the homomorphically encrypted data — confidentiality is not longer warranted.

### 3 Timing Leakage Analysis on Barrett Modular Multiplication

Modular operations can be sped up with some algorithmic techniques. Paul Barrett introduced at the CRYPTO'86 conference a fast algorithm to compute modular operations [3]. This algorithm, as shown in Alg. 4, was tailored for RSA (and happens to work well for Elliptic Curve Cryptography as well)

In the rest of this section, we will analyze a variant of Barrett’s algorithm whereby the modulus size $\ell_q = \lfloor \log_2(q) \rfloor + 1$ can be less than a strict multiple of the machine integer bitwidth $\ell$ (say $\ell = 64$ bits).

---

2The handbook of applied cryptography [14] also provides the same algorithm, in particular the reduction part only, as Alg. 14.42 at page 604 of chapter 14.
Algorithm 4: Barrett Modular Multiplication [3]

Input: Modulus \( q \in \mathbb{N}_{\geq 2} \), \( a, b \in \mathbb{Z}_q \)
Output: \( ab \mod q \)

1. Precompute \( \ell_q = \lfloor \log_2 q \rfloor + 1 \), \( \mu = \lfloor \frac{2^{\ell_q}}{q} \rfloor \)
2. \( z = ab \)
3. \( \tilde{m} = \lfloor \lfloor z/2^{\ell_q-1} \rfloor / 2^{\ell_q+1} \rfloor \)
4. \( r = z - \tilde{m}q \)
5. while \( r \geq q \) do
   6. \( r = r - q \) \hspace{1cm} // There might be 0, 1 or 2 extra-reductions
7. return \( r \)

Remark 1. Note that in the original Barrett modular multiplication as shown in Alg. 4, there might be 0, 1 or 2 extra-reductions as demonstrated in [3,15]. However, in SEAL library, a new variant is implemented. In particular, by aligning with the machine word, \( \ell = 64 \) or 32 bits, that is greater than \( \ell_q \), it results in only 0 or 1 extra-reductions as demonstrated below.

3.1 Barrett Modular Multiplication in SEAL: Principles

The goal is to compute a modular product

\[
ab \mod q
\]

where \( a \) and \( b \) are positive integers, \( 0 < a, b < q \), and where \( q > 2 \) is a fixed (and known) modulus. For that one needs to compute the product \( z = ab \) and then the integer division:

\[
m = \lfloor \frac{z}{q} \rfloor
\]

so that the result is

\[
z \mod q = z - mq.
\]

We assume that

\[
q < 2^\ell
\]

i.e., \( q \) fits at most on \( \ell \) bits (typically \( \ell = 64 \) in SEAL library and \( \ell = 32 \) in its embedded version [16]).

The goal is to replace the integer division \( m = \lfloor \frac{z}{q} \rfloor \) by the fast binary shift corresponding to the integer division by \( 2^\ell \). Let us write

\[
\frac{z}{q} = \frac{z}{2^\ell} \cdot \left( \frac{2^\ell}{q} \right)
\]

where by the assumption \( q < 2^\ell \), the number \( \frac{2^\ell}{q} < 1 \) with \( \ell \) most significant bits given by

\[
\mu = \left\lfloor 2^\ell \cdot \frac{2^\ell}{q} \right\rfloor = \left\lfloor \frac{2^{2\ell}}{q} \right\rfloor \geq 2^\ell
\]
a number (at most $2\ell$ bits) which can be precomputed once and for all in the algorithm. Therefore, in the expression
\[ \frac{z}{q} = \frac{z}{2^\ell} \cdot \left( \frac{2^\ell}{q} \right), \]
we replace the integer division $m = \lfloor \frac{z}{q} \rfloor$ by a lower bound:
\[ \tilde{m} = \left\lfloor \left\lfloor \frac{z}{2^\ell} \cdot \left( \frac{2^\ell}{q} \right) \right\rfloor \right\rfloor = \left\lfloor \frac{z \cdot \mu}{2^\ell} \right\rfloor \]
which requires only two binary shifts $\ell$ bits and a multiplication (by $\mu$) instead of the long division by $q$.

**Remark 2.** As we mentioned in Remark 1, the new variant implemented in SEAL library takes Eqn. (1), that is different from line 3 in Alg. 4. We will show that there shall be at most one extra-reduction, instead of (at most) two extra-reductions in the original Barrett modular multiplication (Alg. 4).

### 3.2 Number of Extra-Reductions

The number of extra reductions (subtractions by $q$) one has to make on $z - \tilde{m}q$ to obtain $z - mq$ equals the error made on the quotient $m - \tilde{m} \geq 0$. That error is of the form
\[ m - \tilde{m} = \left\lfloor \frac{a\lfloor b \rfloor}{c} \right\rfloor - \left\lfloor \frac{a\lfloor b \rfloor}{c} \right\rfloor \]
where we have noted $a = \frac{z}{2^\ell}$, $b = \frac{2^\ell}{q}$, and $c = 2^\ell$. Now excluding the external floor brackets for the moment, we can compute
\[ \Delta = \frac{ab}{c} - \frac{\lfloor a\lfloor b \rfloor \rfloor}{c} = \frac{a\lfloor b \rfloor + \{a\lfloor b \rfloor\}}{c} = \frac{a\lfloor b \rfloor + \{a\lfloor b \rfloor\}}{c} \]
since in the last equality the difference $\{\lfloor a\lfloor b \rfloor \rfloor\} = 0$. That is,
\[ \Delta = \frac{z}{2^\ell} \cdot \left\{ \frac{2^\ell}{q} \right\} + \left\{ \frac{\lfloor \frac{z}{q} \rfloor}{\mu} \right\}. \]  
(2)

The fractional parts $\{\cdot\}$ are evidently $< 1$, so that
\[ \Delta \leq \frac{z}{2^\ell} + \frac{1}{2^\ell} \leq \frac{(2^\ell - 1)^2 + 2^\ell}{2^2\ell} = 1 - \frac{1}{2^\ell} + \frac{1}{2^\ell} < 1 \]
avoidly. Now back to
\[ m - \tilde{m} = \left\lfloor \frac{z}{q} \right\rfloor - \left\lfloor \frac{z}{q} - \Delta \right\rfloor, \]
it is easily seen that, as a general rule if $u < v$, $[v] - [u]$ is equal to the number of whole integers $\in (u, v)$. Thus we have shown:

**Theorem 1.** The number $m - \tilde{m}$ of extra reductions in Barrett’s modular multiplication algorithm equals the number of whole integers lying strictly between $\frac{z}{q} - \Delta$ and $\frac{z}{q}$ where $0 < \Delta < 1$ is given by (2).

In particular, $0 \leq m - \tilde{m} \leq 1$ since there can be at most 1 whole integer in an interval of length $< 1$. 

6
3.3 ℓ-Bit Implementation

Since \( q \) can be any modulus < \( 2^\ell \), \( z = ab \leq (q - 1)^2 \) lies on \( 2\ell \) bits at most. Whenever a number \( x \) is \( 2^{2\ell} \) we break it on two-bit limbs and write \( x = x_1 2^k + x_0 \) (in particular \( x_1 = \left\lfloor \frac{x}{2^\ell} \right\rfloor \). Then \( \mu \) is a constant precomputed as \( \mu = \mu_1 2^\ell + \mu_0 \) and we have

\[
\begin{align*}
  z\mu &= (z_1 2^\ell + z_0)(\mu_1 2^\ell + \mu_0) = z_1\mu_1 2^{2\ell} + (z_1\mu_0 + z_0\mu_1)2^\ell + z_0\mu_0
\end{align*}
\]

so that

\[
\left\lfloor \frac{z \cdot \mu}{2^\ell} \right\rfloor = z_1\mu_1 2^{2\ell} + (z_1\mu_0 + z_0\mu_1) + (z_0\mu_0)_1
\]

and

\[
\left\lfloor \frac{z \cdot \mu}{2^\ell} \right\rfloor = z_1\mu_1 + (z_1\mu_0)_1 + (z_0\mu_1)_1 + \left\lfloor \frac{(z_1\mu_0)_0 + (z_0\mu_1)_0 + (z_0\mu_0)_1}{2^\ell} \right\rfloor \leq 2^\ell
\]

where the addition \((z_1\mu_0)_0 + (z_0\mu_1)_0 + (z_0\mu_0)_1\) (a sum of three ℓ-bit numbers) carries at most \( 1 + 1 = 2 \) at the ℓ-th bit position.

**Remark 3.** By Theorem 1, \( z - \tilde{m}q \) is either \( z \mod q < q \) or \((z \mod q) + q < 2q\). Therefore, if \( q < 2^{\ell-1} \), then \( z - \tilde{m}q < 2^\ell \) can be computed as

\[
z - \tilde{m}q = z_0 - (\tilde{m}q)_0.
\]

In a C implementation of unsigned ℓ-bit type (e.g. \( \ell = 64 \) unsigned long long), what overflows \( 2^\ell \) in the product \( \tilde{m}q \) is automatically discarded so that one can then write \( z_0 - \tilde{m}q \) in place of \( z_0 - (\tilde{m}q)_0 \).

3.4 Timing Leakage Analysis

Assuming that \( u = \frac{z}{q} \) can be “any” arbitrary number in the acceptable range, the probability that one extra-reduction occurs is the probability \( P_\Delta \) that the arbitrary interval \((\frac{z}{q} - \Delta, \frac{z}{q})\) of length \( \Delta \) contains exactly one whole integer. It is “obviously” equal to \( \Delta \) (see following detailed proof).

**Proof.** For fixed \( \Delta \in (0,1) \), the interval \((u - \Delta, u)\) contains the whole integer \( n \) (which is necessarily \( \lfloor u \rfloor \)) if and only if \( u - \Delta < n < u \), that is, \( n < u < n + \Delta \). This probability is independent of \( n \) and we may assume that \( n = 0 \) so that \( u \) is uniformly distributed in \([0,1]\). Therefore, the probability of one extra-reduction given \( \Delta \) is the probability that \( u < \Delta \), which is \( P_\Delta = \Delta \).

Now the fractional part of \( \frac{2^\ell}{q} \):

\[
\nu = \left\{ \frac{2^\ell}{q} \right\}
\]

in (2) is known, however the term \( \left\{ \frac{z}{q} \right\} \cdot \mu \) is difficult to locate between 0 and 1. It follows that

\[
\frac{\nu}{2^\ell} z < \Delta = P_\Delta \leq \frac{\nu}{2^\ell} z + \frac{1}{2^\ell}
\]

In particular

\[
P_\Delta \leq \frac{\nu (q - 1)^2}{2^\ell} + \frac{1}{2^\ell}
\]

(3)
Since in practice $q$ is chosen quite small w.r.t. $2^\ell$, $\mathbb{P}_\Delta$ is very very small as the following examples show:

**Example 1** (Case of long online measurement, fast cryptanalysis). Consider $q = 132120577$ (so $(q-1)^2 = 17455846602571776$), $\ell = 64$ (so $2^{-\ell} \approx 5.42 \cdot 10^{-20}$):

$$\nu = \left\{ \frac{2^{2\ell}}{q} \right\} = \frac{42396112}{132120577} \approx 0.32$$

and

$$\nu(q-1)^2 \approx 1.65 \cdot 10^{-23}$$

so essentially

$$\mathbb{P}_\Delta \lesssim 5.42 \cdot 10^{-20}$$

It can be seen that the probability of an extra-reduction is very small, though nonzero. Therefore, a fair amount of time is required to find a multiplication which creates an extra-reduction. However, as we shall see in Sec. 4.3, the offline cryptanalysis will be fast and conclusive with the knowledge of a few such ciphertexts $c[1]$ (as few as one).

Besides, it is also possible to make a tradeoff. As shown in the example below, the probability is less, hence more values of $c[1]$ leading to an extra-reduction in the Barrett algorithm can be found. The downside is that the cryptanalysis will take longer (from a computational complexity standpoint).

**Example 2** (Case of faster online measurement, slower cryptanalysis). In this case, we take $q = 18014398492704769$ (so $(q-1)^2 = 32451855305396381725731640993824$), $\ell = 64$ (so $2^{-\ell} \approx 5.42 \cdot 10^{-20}$):

$$\left\{ \frac{2^{2\ell}}{q} \right\} = \frac{17979488999555073}{18014398492704769} \approx 0.998$$

and

$$\nu(q-1)^2 \approx 9.52 \cdot 10^{-7}$$

so essentially

$$\mathbb{P}_\Delta \lesssim 9.52 \cdot 10^{-7}.$$
therefore,

\[
\frac{\nu(q - 1)^2}{2^\ell} \approx 4.54 \cdot 10^{-2}
\]

which resulting in

\[
P_\Delta \lesssim 4.54 \cdot 10^{-2}.
\]

As we will demonstrate in the next section, this probability \( P_\Delta \) is easy to achieve in practice and the corresponding attack shall be very efficient.

4 Cache-Timing Attacks

4.1 Principle

Cache-timing attack is a well-known class of side-channel attacks that attempt to gain information about which memory locations certain victim programs access. The response time of cache hit and cache miss are different. Attackers can infer the information in the cache through the difference of access time, so as to obtain secret data. Common cache-timing attacks can be divided into three categories: Prime+Probe, Flush+Reload, and Evict+Time.

Prime+Probe [19] is the oldest and most common cache attack. This attack targets a single cache set and detects any access to any address in that cache set by victim program. The initialization phase of its active portion is called "prime". In this phase, the attacker accesses enough cache lines from the cache set to completely fill the cache set with their own data. Then, during the measurement phase, named "probe", the attacker reloads the same data that was primed before and calculates the time cost of this operation. If the victim does not have access to the data in the target cache set, this operation will proceed quickly. Conversely, if the victim accesses data in the target cache set, that access will evict a portion of the primed data, resulting in slower reload due to additional cache misses. Therefore, a slow measurement phase means that the victim accessed data in the target cache set.

Flush+Reload [26] relies on shared memory. It targets the specific cache line and detects any other program’s access to that cache line. This makes Flush+Reload a more precise attack than Prime+Probe. Due to the shared L3 cache, Flush+Reload also works naturally across cores. As with all L3 cache attacks, Flush+Reload can detect accesses to instructions or data. An improved variant of Flush+Reload, Flush+Flush [11], is based on the fact that CLFLUSH instructions execute differently in different cache states. If the target of CLFLUSH exists in the cache, it needs to be evicted from the multi-level cache during the execution of the CLFLUSH instruction, so the execution time is longer. Otherwise, the execution time will be shortened. The difference between the execution times of the instruction allows the attacker to determine whether the cache line has been accessed by the victim program. Note that Flush+Flush utilizes instruction execution time rather than memory access, so it is prone to false positives and false negatives.

In Evict+Time [18], the attacker first lets the victim program run normally, and records the time to establish the baseline. Next, the attacker will evict some cache lines and let the victim program run again. By comparing the victim program’s time to the baseline, an attacker can tell if the victim program is using an evicted cache line. It should be noted that Evict+Time needs to be executed multiple times. If the victim program is launched only once, accurate results cannot be obtained.
Cache-Timing Attack on the SEAL Homomorphic Encryption Library

4.2 Cache-timing attack on SEAL

Side-channel analyses on FHE have already been published (see e.g., [2]). Typically, such attack must be carried out locally with a side-channel probe. We propose an attack which can be executed remotely. In fact, regarding SEAL library, the attack aims to find the secret key used in the function of dyadic_product_coeffmod (line 226 in polyarithsmallmod.cpp), where the Barrett reduction is performed (line 259) to compute the modular product \( c \times s_k \mod q \). It corresponds to line 1 of Alg. 3.

In this work, we focus on a cache-timing attack which can be realized from the remote, as explained in Sec. 4.1. We target the function barrett_reduce_128 implemented at line 166 in SEAL/native/src/seal/util/uintarithsmallmod.h in SEAL library. This function implements the conditional extra-reduction in a non-constant manner. Indeed, it leverages the ternary operator \(?:\) in the macro SEAL_COND_SELECT below:

Listing 1: Exemplary code segment for Barrett algorithm in SEAL.

```
// Barrett subtraction
    tmp3 = input[0] - tmp1 * modulus.value();

// One more extra-reduction (subtraction) is enough
    return SEAL_COND_SELECT(tmp3 >= modulus.value(), tmp3 - modulus.value(), tmp3);
```

This timing leakage has been detected “manually”, by code reading, because the state-of-the-art automated detection tools require C code [22] or machine code [17]. Here, the code of SEAL is implementing inline template functions, whose mapping with compiled code is not easy.

4.3 Key-Recovery Results

The statistics and exploitation of cache-timing attacks on Barrett multiplication has already been studied by Mittmann & Schindler in [15], albeit on the variant of Barrett algorithm shown in Alg. 4. The behavior in terms of existence of an extra-reduction is different for the variant of Barrett multiplication implemented in SEAL. We conduct thereafter the analysis of the way the extra-reduction does relate to the secret key \( s_k \).

Let us assume that we know a pair \((c,\text{red})\in\{1,\ldots,q\}\times\{0,1\}\), such that:

\[
\forall 1 \leq i \leq Q, \quad \left( \left\lfloor \frac{c_i \times s_k}{q} \right\rfloor - \left\lfloor \frac{c_i \times s_k}{2^\ell q} \right\rfloor \right) = \text{red}_i
\]

where \( s_k \in \{1,\ldots,q-1\} \) is the unknown.

Let us keep only the cases where there is an extra-reduction (described in Sec. 3.4). When, we know that: \( z/q \) is close to an integer (at distance \( \max \Delta \)). Then, for a subset of indices \( i \in \{1,\ldots,Q\} \), we have

\( (c_i \times s_k \mod q) \in \{1,\ldots,\epsilon q\} \),

where \( \epsilon (= \Delta) \) is small. That is, we have, for some \( m_i \in \mathbb{Z} \):

\[
c_i \times s_k + m_i \times q = r_i, \quad \text{where } r_i \in \{1,\ldots,\epsilon q\}.
\]
For each $r_i$, we can solve the Diophantine equation (4) for $s_k$ and $m_i$ (because $c_i$ and $q$ are coprime). Namely, since $c_i$ and $q$ are coprime, we can determine $(u, v)$ such that:
\[ c_i \times u + q \times v = 1. \] (Bézout theorem)
Thus one solution is
\[ s_{k0} = ur_i \quad \text{and} \quad m_{i0} = vr_i. \]
We know that all the solutions are:
\[ s_k = s_{k0} - \lambda q \quad \text{and} \quad m_i = m_{i0} + \lambda c_i, \]
where $\lambda$ is an arbitrary integer. Since $s_k$ lives in $\mathbb{Z}_q$, we only keep $s_{k0} - \lfloor s_{k0}/q \rfloor$ as a solution (hence $\lambda = \lfloor s_{k0}/q \rfloor$).

The process is captured in Alg. 5. If the algorithm returns multiple key candidates (list $SK$ is not a singleton), then the algorithm is re-executed on another pair $(c, red)$, and the new list of key candidates is the intersection of the two key lists. This pruning strategy is repeated until only one single key remains.

**Algorithm 5: Key-Recovery Cryptanalysis**

**Input:** $c \in \{1, \ldots, q-1\}$ such that there is an extra-reduction in the Barrett multiplication with a constant $s_k$.

**Output:** List $SK$ of possible secret keys value $s_k$.

1. $SK \leftarrow \emptyset$ // Empty list
2. $(u, v) \leftarrow \text{Bézout}(c, q)$, based on extended Euclid algorithm // $uc + vq = 1$
3. for $r \in \{1, \ldots, \Delta q\}$ do // For $\Delta$, use the upper bound given in (3)
4. \[ s_{k0} \leftarrow r \times u; \quad m_0 \leftarrow r \times v \] // $c \times s_{k0} + q \times m_0 = r$
5. \[ \lambda = \lfloor s_{k0}/q \rfloor \]
6. $SK \leftarrow SK \cup \{s_{k0} - \lambda q\}$
7. return $SK$

Namely, the set of possible secret keys $\hat{s}_k$ can be characterized by and further refined as per:

\[
\hat{s}_k = \bigcap_{i=1}^{Q} \{ x \in \mathbb{F}_q \mid c_i x \leq \Delta q \} \quad (5)
\]
\[
= \{ x \in \mathbb{F}_q \mid \forall i, 1 \leq i \leq Q, c_i x \leq \Delta q \} \quad (6)
\]
\[
= \left\{ \frac{r}{c_1} \mid r \in \mathbb{F}_q \text{ and } \forall i, 1 \leq i \leq Q, r\frac{c_i}{c_1} \leq \Delta q \right\} \quad (7)
\]
\[
= \left\{ \frac{r}{c_1} \mid r \in \{1, \ldots, \Delta q\} \text{ and } \forall j, 2 \leq j \leq Q, r\tilde{c}_j \leq \Delta q \right\}. \quad (8)
\]

In this series of equivalent distinguishers, we leveraged:

- variable change $r = c_1 x$ between lines 6 and 7,
- condition that $r\frac{c_i}{c_1} \leq \Delta q$ is equivalent to $1 \leq r \leq |\Delta q|$ when $i = 1$ between lines 7 and 8,
- precomputation of reduced ciphertexts $\tilde{c}_j$ (with respect to pivot ciphertext $c_1$), namely $
\tilde{c}_j = \frac{r}{c_1}$. 

The complexity of the computation is reduced from line to line, namely it is:

- $O(qQ)$ at line 5,
- $O(q)$ at lines 6 and 7,
- $O(\Delta q)$ at line 8.

Therefore the overall complexity of our cryptanalysis is $O(\Delta q)$. Interestingly, it does not depend on $Q$.

The attack results are presented graphically in Fig. 1. This figure represents, in logarithmic scale, the number of candidates for $s_k$, as a function of the number $Q$ of ciphertexts which cause an extra-reduction. It also represents, in linear scale, the time taken for the off-line part of the attack (in seconds) coded in Magma [24], running on a quadcore with Intel(R) Xeon(R) CPUs cadenced at 2.0 GHz, using 16 GBytes of RAM.

In this figure:

- the values for 0 ciphertext are trivial: all $q$ keys are possible and it require no time to make such useless deduction. Still, this value allows to materialize the starting point of the attack;
- the values for 1 ciphertext correspond to the case where Alg. 5 is executed once, hence our optimization (equation (8)) is not leveraged. The optimization starts to be useful from 2 ciphertexts on; indeed, the attack activates shortcuts in the enumeration within the interval of size $\Delta q$ while at the same time pruning many key candidates.

![Figure 1: Performance of the attack, leveraging distinguisher Eqn. (8)](image)

The number of solutions after $Q = 1, 2, 3,$ etc. ciphertexts is represented in Tab. 1. In this case, the number of possible candidates for $s_k$ is dramatically reduced when the number of ciphertexts increases, particularly, the secret key $s_k$ is uniquely determined after $Q = 8$ ciphertexts.
Table 1: Number of solutions for the cryptanalysis in recovering the secret key $s_k$ when the number of ciphertexts increases.

<table>
<thead>
<tr>
<th>Number of c</th>
<th>Number of solutions for $s_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48220170</td>
</tr>
<tr>
<td>2</td>
<td>2188334</td>
</tr>
<tr>
<td>3</td>
<td>99295</td>
</tr>
<tr>
<td>4</td>
<td>4508</td>
</tr>
<tr>
<td>5</td>
<td>192</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

In this experiment, the upper bound on $\Delta$ is 0.045. This is reflected by the geometrical decrease of the number of solutions by a rate $\approx \Delta$ in Tab. 1.

**Remark 4** (Computation optimization of Alg. 5). *For each ciphertext $c_i$ causing an extra-reduction, a large loop over $\Delta q$ possible values of $r$ must be executed (see line 3). It is possible to precompute a pruned list of such $r$s by noting that, at the end of the process, we have that all $r_i/c_i \mod q$ are the same (namely equal to $s_k$). Therefore, it is possible to select the values of $r$ corresponding to ciphertext $c_i$ such that all $r_i c_i / c_j$ for $j \neq i$ also meet the requirement $r_i c_i / c_j \in \{1, \ldots, \Delta q\}$.

### 4.4 Countermeasure

Essentially, the timing leakage caused by the extra-reduction is the non-constant time implementation of Barrett modular multiplication as shown in Listing 1. Therefore, this timing leakage shall be removed by implementing the macro `SEAL_COND_SELECT` in a constant time manner.

The improved macro is shown in Listing 2, in which the well-known trick is applied to transform the ternary condition operator into non-conditional implementation. In particular, the variable `val` will equal the modular value $q$ if `falg` is true, and 0 otherwise. Therefore, the following constant time implementation is equivalent to the one in Listing 1.

Listing 2: Constant time implementation of the macro in SEAL library.

```c
// Barrett subtraction
tmp3 = input[0] - tmp1 * modulus.value();

flag = (tmp3 >= modulus.value()) // flag must be an unsigned integer
val = (!flag) & modulus.value();

return tmp3 - val;
```

### 5 Conclusion

Barrett multiplication has been one of the common techniques to implement fast modular multiplication like in SEAL FHE library. In this work, we target this elementary module, especially
the new variant implemented in SEAL library, and exhibit a cache-timing vulnerability that exploits the existence of extra-reductions as side-channel leakage. We present a key-recovery attack that utilizes this leakage, which can determine the secret key with a few ciphertexts. Our simulated experimental results confirm our findings.

As perspective, we will explore the practical applications of our attack with cache-timing leakage. Furthermore, as a fundamental block, Barrett modular multiplication must be protected against such attacks that exploit the timing leakage of extra-reductions. For applications, we will apply our findings to lattice-based post-quantum cryptography schemes like Crystals-Kyber and Crystals-Dilithium with different sets of parameters, where our cryptanalysis shall be very efficient with small modulus values.

References

Cache-Timing Attack on the SEAL Homomorphic Encryption Library

Cheng et al.


