D5.7 Final Report on Hardware-Operated Schemes

Document Identification			
Status	Final	**Due Date**	30/09/2020
Version	0.1	**Submission Date**	30/09/2020

Related WP	WP5	**Document Reference**	D5.7
Related Deliverable(s)	D5.1, D5.2, D5.4, D5.6, D5.8	**Dissemination Level(*)**	CO
Lead Participant	KU Leuven	**Lead Author**	Angshuman Karmakar
Contributors	KU Leuven, UH	**Reviewers**	Kimmo Järvinen (UH) ATOS FUAS

Keywords:
Functional encryption, physical adversary, side-channel attacks, countermeasures, lattice-based cryptography

This document is issued within the frame and for the purpose of the FENTEC project. This project has received funding from the European Union's Horizon2020 under Grant Agreement No. 780108. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level.-PU: Public, fully open, e.g. web; CO: Confidential, restricted under conditions set out in Model Grant Agreement; CI: Classified, Int = Internal Working Document, information as referred to in Commission Decision 2001/844/EC.

Document Information

List of Contributors	
Name	Partner
Josep Balasch	KU Leuven
Angshuman Karmakar	KU Leuven
Jose Maria Bermudo Mera	KU Leuven
Svetla Nikova	KU Leuven
Kimmo Järvinen	UH

Document History			
Version	Date	Change editors	Changes
0.1	20/08/2020	Angshuman Karmakar (KU Leuven)	ToC
0.2	16/09/2020	Angshuman Karmakar (KU Leuven)	Added Chapter. 5
0.3	15/09/2020	Jose Maria Bermudo (KU Leuven)	Added Chapter. 4.4
0.4	20/09/2020	Angshuman Karmakar (KU Leuven)	Version for internal review
0.5	22/07/2020	Angshuman Karmakar (KU Leuven)	Edits internal review by Kimmo Järvinen (UH)
0.6	28/09/2020	Angshuman Karmakar (KU Leuven)	Edits internal review by Angshuman Karmakar (KUL)
1.0	28/09/2020	Angshuman Karmakar (KU Leuven)	Final version, edits based on feedback by Angshuman Karmakar (KUL)

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Angshuman Karmakar (KU Leuven)	30/09/2020
Technical Manager	Michel Abdalla (ENS)	30/09/2020
Quality Manager	Diego Esteban (ATOS)	30/09/2020
Project Coordinator	Francisco Gala (ATOS)	30/09/2020

Contents

List of Figures

List of Acronyms

Abbreviation / acronym	Description
ALU	Arithmetic Logic Unit
BRAM	Block Random-Access Memory
CDT	Cumulative Distribution Table
CRT	Chinese Remainder Theorem
DDG	Discrete Distribution Generation
DMA	Direct Memory Access
DSP	Digital Signal Processing
EM	Electro-Magnetic
FE	Functional Encryption
FF	Flip-flop
HW	Hardware
KEM	Key Encapsulation Method
LSB	Least Significant Bit
LUT	Look-up Table
LWE	Learning With Errors
LWR	Learning With Rounding
NP	Non-deterministic Polynomial time
NTT	Number Theoretic Transform
PQ	Post Quantum
RLWE	Ring Learning With Errors
RNS	Residual Number System
SIMD	Single Instruction Multiple Data
SRAM	Static Random-Access Memory
SW	Software

Executive Summary

In this deliverable D5.7 “Final Report on Hardware-Operated Schemes” we describe the work that has been done in FENTEC related to the development of hardware-operated schemes. The deliverable summarizes activities in Task 5.4 until September 2020. As defined in D5.1 “Security and Trust Models” the adversarial model involved in hardware-operated schemes is that of an adversary who is capable to mount physical attacks on FE implementations. Note that in contrast to hardware-assisted schemes (covered in D5.4), the complete platform is considered untrusted. Thus any part of the implementation, whether running on SW and/or HW, needs to incorporate built-in protections against physical attacks.

From the computational point of view, lattice-based cryptosystems has two major components polynomial multiplication and noise sampling. In this final deliverable we present two techniques that serve to protect these two essential building blocks for lattice-based cryptosystems, against timing attacks. For each of them, we develop and instantiate novel methods that ensure constant execution time and provide results to benchmark their performance. We proposed an architecture with a HW-SW co-design approach for polynomial multiplication. We also present a functional encryption scheme based on ring learning with errors hard problem. Cryptosystems based on ring learning with errors are more efficient than the cryptosystems based on standard lattices due to the use of faster polynomial arithmetic. We describe several parameters choices for this scheme. Finally, we describe an efficient approach to multiply large polynomials used in our scheme.

1 Introduction

1.1 Purpose of the document

Deliverable D5.7 "Final Report on Hardware-Operated Schemes" gives a description of work in WP5 of FENTEC and, in particular, in Task 5.4 until September 2020. The deliverable provides details about research on HW-operated schemes for FE. In contrast to Tasks 5.2 and 5.3, the target platform in Task 5.4 is assumed to be fully untrusted and therefore all components are susceptible to physical attacks.

The goal of the research activities on HW-operated schemes is to provide FE implementations that incorporate resistance against physical attacks. The work performed so far in Task 5.4 has focused on implementations resistant to timing side-channel attacks, that is, involving an adversary that attempts to exploit variabilities in execution time. The natural countermeasure against such an attacker is to devise regular algorithms that run in constant time, while minimizing performance losses. In this deliverable we present methods to protect essential components of lattice-based cryptography from side-channel attacks, a FE scheme based on ring learning with errors, and implementations of building blocks for FE lattice-based schemes.

1.2 Structure of the document

This deliverable is structured as follows. Chapter 2 revisits the trust models defined in D5.1 with a focus on the HW-operated trust model. It also presents the adversarial model considered in the work performed so far in Task 5.1, that is, a side-channel adversary that exploits timing leakages. In Chapter 3 we present a secure method to sample from a Gaussian distribution, while in Chapter 4 we introduce a technique to perform fast and time-constant polynomial multiplication. We also describe a HW-SW co-design architecture for efficient polynomial multiplications. Chapter 5 describes our FE scheme based on the ring learning with errors, parameters, and an efficient technique to multiply polynomials to be used in the scheme. Chapter 6 ends the deliverable with conclusions and outlines some ideas for future work.

2 Trust model

The deliverable D5.1 defined three different trust models for computing platforms that are considered in FENTEC. Figure 1 shows these trust models as a recap from D5.1. This deliverable focuses on HW-operated schemes shown in the right-hand side and, thus, relates mainly to the work done in Task 5.4 of WP5.

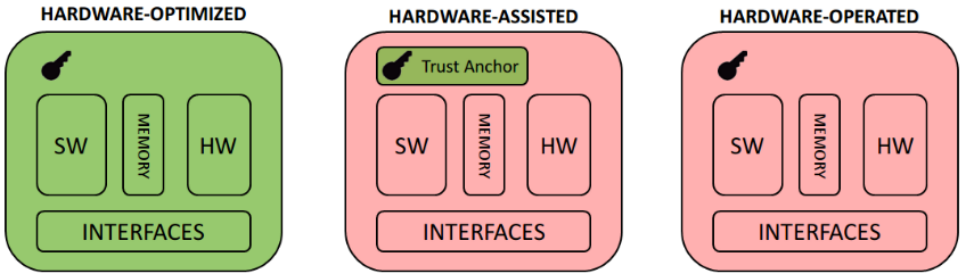

Figure 1: Underlying security models for the different tasks in WP5. Green (trusted environment), red (untrusted environment).

The trust model of HW-operated schemes assumes no trust in the computing system. This corresponds to the strongest adversarial model in terms of physical security. It involves an adversary with physical access to a target platform that performs sensitive cryptographic operations. Note that in contrast to the HW-assisted trust model, none of the platform components can be considered to be tamper-proof. Consequently, all building blocks in the FE implementation, whether running in SW or in HW, need to provide some level of resistance against physical attacks.

As introduced in D5.1, physical attacks target crytographic implementations on computing platforms with the aim of extracting secret cryptographic keys. They typically assume the adversary to be in physical possession of the device, e.g. an edge node operating in the field. Physical attacks exploit information gained by observing or manipulating physical and accessible channels. Depending on the role of the adversary, passive or active, attacks are often categorized in two families:

- Side-channel attacks (SCA) collect and exploit information that leaks through physical channels during normal operation of the target platform. Typical examples include execution time [29], power consumption [30], and electromagnetic emanations [21]. All these physical phenomena carry information about internal operations executed by the platform, and their analysis may therefore enable recovery of sensitive keys processed by the implementation.
- Fault attacks (FA) exploit intentional errors induced to a computing platform through physical channels. Many techniques can be used by adversaries to inject faults, for instance, clock glitches [8], electromagnetic pulses [43] or lasers [48]. The insertion of errors during cryptographic computations can disrupt the expected program flows (e.g. skipping critical instructions) or the expected data flow (e.g. changing the value of intermediate results).

Adversary model

In this intermediate deliverable we focus on implementing building blocks for FE schemes that are immune to timing attacks. Variabilities in the execution time of an implementation represent

the most essential type of side-channel leakage that an adversary can exploit. It is therefore a natural starting point for research on hardware-operated schemes, as further countermeasures (e.g. against power and/or electromagnetic side-channel attacks) are built on top of time-constant implementations.

In what follows, we consider an adversary that can accurately measure the execution time of an implementation, e.g. with the aim of recovering secret keys due to timing differences. Timing side-channel attacks were first documented by Kocher at Crypto'96 [29], which showed how data-dependent variances in the execution time of public-key cryptographic implementations (among others, RSA, DSS and Diffie-Hellman) could allow an adversary to infer the secret key bits in a divide-and-conquer fashion. Since this seminal work, many attacks have appeared targeting popular libraries such as OpenSSL (e.g. RSA attack by Brumley and Boneh [13]) or widely extended systems such as TLS [3]. In parallel, cache-timing side-channel attacks [39, 9] have risen as a popular attack path against cryptographic libraries. They exploit the fact that caches in modern processors influence the memory access patterns of programs, which may reveal sensitive information when they depend on secret data.

As illustrated by the amount of attacks disclosed in recent years, preventing timing side-channel attacks is not a trivial task. One needs to start from a regular algorithm specification, with no data-dependent execution flows, and to avoid the use of memory lookups that depend on sensitive data. While conceptually simple, enforcing these conditions is not trivial. In addition, constant-time implementations cannot contain optimizations for frequent and/or corner cases. Consequently, they often incur significant performance overheads that have to be minimized.

In the rest of this deliverable, we present two examples on how to design and implement constant-time operations found at the core of lattice-based FE cryptosystems. Both implementations are not only designed to ensure constant execution time, but also to be competitive in terms of performance. Note that, in addition to design and implementation, it is also critical to experimentally evaluate the security of building blocks. This aspect is however part of Task 5.5, and will therefore be documented in D5.8.

3 Constant-time Gaussian sampler

Most cryptosystems based on lattice problems, such as LWE or RLWE, require to generate an *error* term to hide secret keys. Traditionally, these error terms are sampled from a Gaussian distribution which itself is not trivial to implement. A lot of effort has been devoted to devise efficient methods to generate Gaussian samples, see for instance [38, 19, 18]. However, the sampling process in these methods runs in non-constant time which opens up new avenues for side-channel attacks. In order to cope with this, a current trend is to generate samples from other distributions, e.g. binomial [4, 12] or uniform [14, 16], from which it is easy to generate samples in constant-time. The security of the cryptosystem is then evaluated by modeling these distributions as Gaussian distributions. While this approach works well for certain cryptographic constructions, it may lead to larger key sizes and costlier computations. Therefore constant-time Gaussian samplers are essential to build up side-channel secure FE implementations.

Techniques to mitigate side-channel leakage from Gaussian samplers have been recently proposed in the literature, but they either sacrifice efficiency [11] for constant-time operation or do not provide adequate security [45]. The work in [25] introduced a constant-time Gaussian sampler based on evaluation of Boolean expressions and the usage of bit-slicing techniques. The work did however not provide concrete details on how to generate these Boolean functions.

In this Chapter, we present efficiency improvements to the work of [25] by showing how to create Boolean functions that map random bits to the samples of an arbitrary discrete Gaussian function. By carefully analyzing the Knuth-Yao [27] sampling for discrete Gaussian we observe a special property of the input random strings that generate samples. We show how to leverage this property to minimize the Boolean functions efficiently and show it can speed up the sampling by up to 37% when compared to the minimization technique in [25].

The contents of this Chapter are extracted from the following FENTEC publication:

A. Karmakar, S. Sinha Roy, F. Vercauteren, I. Verbauwhede: *Pushing the speed limit of constant-time discrete Gaussian sampling. A case study on the Falcon signature scheme.* DAC 2019: 88:1-88:6

In what follows, we provide a brief description of our method and performance results on an x86-64 processor that serve to compare with the work in [25]. For detailed technical information, we refer the reader to the complete work in [26].

3.1 Preliminaries

In this section, we define the different notations used throughout this work. We briefly describe the discrete Gaussian distribution and the Knuth-Yao sampling which is used to generate samples from discrete Gaussian distributions.

We define $\mathbb{Z}^* = \{0\} \cup \mathbb{Z}^+$. All the binary strings are evaluated in the reverse order, *i.e* the binary evaluation of an n-bit string $b = b_{n-1}b_{n-2}\cdots b_1 b_0$ with b_0 being the LSB, is $2^{n-1}\cdot b_0 + 2^{n-2}\cdot b_1 + \cdots + 2\cdot b_{n-2} + b_{n-1}$. Boolean `and`, `or`, and `not` operations are denoted by the symbols &, |, $^-$ respectively. The terms Boolean function and Boolean expression are used interchangeably. We use f_η to denote Boolean functions that map η Boolean variables to a single Boolean variable. If b is a Boolean variable then we denote repetition of b, i times by b^i. Also, we denote a Boolean string of length i where each variable is either 0 or 1 by $(0/1)^i$.

For binary trees, we denote the level where children of the root exist as the 0-th level. Children of these nodes in turn reside at the 1-st level, and so on. Hence, starting from root we need $i+1$ *steps* to reach nodes at level i. We assume that the standard deviations in our sampler are *small*, and our sampler can be used as a base sampler in systems where samples from discrete Gaussian distribution with large standard deviation are generated by combining samples from discrete Gaussian distribution with small standard deviation. We also use σ to denote the standard deviation of a Gaussian distribution.

3.1.1 Discrete Gaussian distribution

The probability distribution function of a discrete Gaussian distribution is given as

$$\mathcal{D}_\sigma(X = z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-c)^2/2\sigma^2}.$$

Here, X is a random variable defined over $\mathbb{Z}$. In this work, we always consider discrete Gaussian distributions that are centered around 0, i.e. with $c = 0$. Due to the symmetry of the probability density function, it is sufficient to generate samples over $\mathbb{Z}^+$ and use a random bit to determine the sign. For most practical scenarios, all samples are generated in the interval $[0, \tau\sigma]$, where τ is a positive constant known as *tail-cut* factor. Again for practical reasons, the probabilities $\mathcal{D}_\sigma(x)$ are calculated only up to n-bit precision. We denote this as $\mathcal{D}^n_\sigma(x)$.

3.1.2 Knuth-Yao sampling

Dwarakanath and Galbraith [20] first discoursed on the idea of using the well known Knuth-Yao [27] sampling method to generate samples from discrete Gaussian distributions. This method can be divided into two stages. In the precomputation stage, for a particular standard deviation, this method first creates a probability matrix of dimension $(\tau\sigma + 1) \times n$ where a row consists of $\mathcal{D}^n_\sigma(v)$ if $v = 0$ and $2 \cdot \mathcal{D}^n_\sigma(v)$ for all other $v \in [1, \tau\sigma]$. Using this probability matrix a binary tree named discrete distribution generation (DDG) tree is created such that: the Hamming weight of the i-th column of the probability matrix equals the number of leaf nodes in the i-th level of the tree, and each leaf node contains a sample value in the sample space $[0, \tau\sigma]$. An example is shown in Fig. 2.

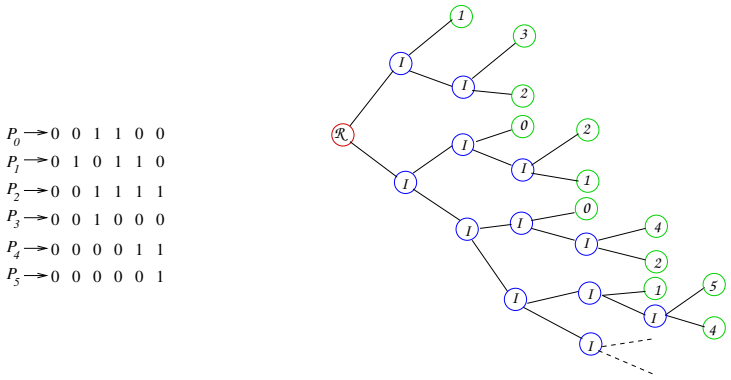

Figure 2: Probability matrix and corresponding DDG tree for $\sigma = 2$ and $n = 6$. Red, blue and green nodes denote the root, intermediate and leaf nodes respectively.

During sampling, a random walk is started from the root node using random bits at every step to decide between the bottom or top subtree. The sampling stops when this random walk hits

a leaf node and the sample value associated with the leaf node is returned as sample. It is worthwhile to note here that for a distribution $\mathcal{D}$ with finite support s.t. $\sum_{x \in Supp(\mathcal{D})} \mathrm{P}^n(x) = 1$ where $\mathrm{P}^n(x)$ is the probability of x up to n-bit floating point precision under the distribution $\mathcal{D}$, the DDG tree is finite and it is possible to generate samples that follow the distribution $\mathcal{D}$ *exactly*. In contrast, for distributions like discrete Gaussian distributions with infinite support the DDG tree grows infinitely and it is not possible to generate samples that follow the discrete Gaussian distribution *exactly*. In this case, the τ and n are chosen such that the statistical distance between the generated distribution and the actual distribution is lower than $2^{-\lambda}$, where λ is a security parameter.

In the following section, we describe an algorithm to generate samples from a discrete Gaussian distribution using the Knuth-Yao sampling efficiently.

3.1.3 Column scanning Knuth-Yao sampling

The column scanning Knuth-Yao sampling algorithm from [47] generates the DDG tree on-the-fly, thus making the sampling process time and memory efficient. This is shown in Alg. 1. We denote the Hamming weight of column i of the probability matrix P as $\mathrm{h_i}$. Define, $\mathrm{GAP^i}$ as,

$$\mathrm{GAP^i} = (b_0 \cdot 2^i + b_1 \cdot 2^{i-1} + \cdots + b_i) - (\mathrm{h}_0 \cdot 2^i + \mathrm{h}_1 \cdot 2^{i-1} + \cdots + \mathrm{h_i}) \tag{1}$$

here b_j is the output of *RandomBit()* during the j-th iteration of the **while** loop in Alg. 1. It is evident from the above algorithm that a sample is found in the i-th column if and only if $\mathrm{GAP^i} < 0$ and $\mathrm{GAP^{i'}} \geq 0, 0 \leq i' < i$.

Algorithm 1: Knuth-Yao column scanning Sampling

input : Probability matrix P
output: Sample value s

```
1  d ← 0;// Distance between the visited and the rightmost internal node
2  Hit ← 0;// 1 when sampling process hits a terminal node
3  col ← 0;// column number of probability matrix
4  while Hit=0 do
5      r ← RandomBit();
6      d ← 2 * d + r;
7      for row=MAXROW down to 0 do
8          d ← d − P[row][col];
9          if d=-1 then
10             s ← row;
11             Hit ← 1;
12             ExitForLoop();
13     col ← col + 1;
14 return s
```

3.1.4 Bit-sliced Gaussian sampler

The key observation in the bit-sliced discrete Gaussian proposed in [25] was that there exists a unique path from root to each leaf of the DDG tree, since it is a binary tree. This path is

determined by the input random bits to the sampler. This implies that there exists a many-to-one mapping between the set of input random bit strings $(b_0 b_1 \cdots b_{n-1})$ to the set of sample bits. Further, this mapping can be expressed as a set of Boolean functions f_n^ι, $\iota \in [0, m-1]$, where m is the maximum possible bit length of any sample. This is shown in Fig. 3. Now, each sample bit can be calculated in constant-time by executing the corresponding Boolean function completely.

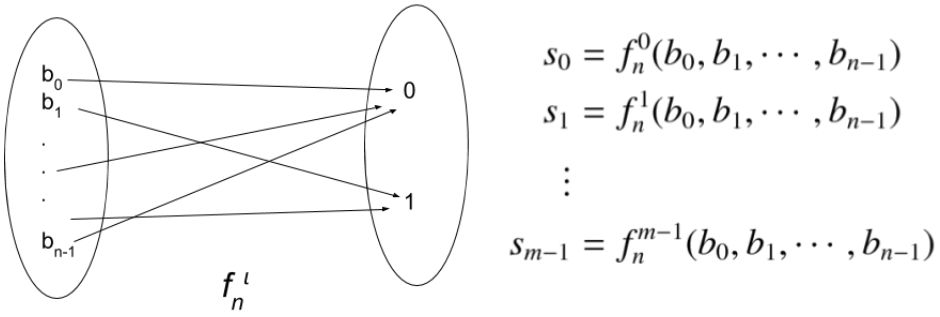

Figure 3: Mapping n random bits to output sample bits and corresponding Boolean functions.

The above method for sampling, though being a constant-time algorithm, has very poor efficiency. So, the next key observation was that modern processors have wide word lengths and bit wise Boolean operators which can be exploited to generate multiple samples in a batch in a single instruction multiple data (SIMD) fashion. In this method, assuming word length w, a variable $b_i^{var}, i \in [0, n-1]$ is packed with w input random bits $b_i^0, b_i^1, \cdots, b_i^{w-1}$. These random variables are then used to evaluate Boolean function f_n^ι using bitwise Boolean operators to generate variables $s_\iota^{var}, \iota \in [0, m, -1]$ which are packed with w output sample bits $s_\iota^0, s_\iota^1, \cdots, s_\iota^{w-1}$. These variables are then unpacked to generate w samples at a time. Despite the overhead of packing and unpacking bits, this method was shown to be approximately two times faster (for $\sigma = 6.15543$) than the best known alternative for constant-time sampling *i.e* a cumulative distribution table (CDT) based sampling with linear search [11].

3.2 Our method

In this section we describe our Boolean minimization strategy for discrete Gaussian sampling. We consider that the Knuth-Yao sampler always takes input random bit strings of length n. However, it is possible that a sample is found on the c-th level of the DDG tree or equivalently the sampler needs only $c+1$ bits to generate the sample. The remaining $n-(c+1)$ bits do not influence the outcome of the sampling. We call them *don't care*(x) bits.

Our method builds on Theorem 1, which establishes an important property of the structure of the input random bit strings which generate the samples. We recall this theorem below and refer to the original work in [26] for the proof.

Theorem 1. *All the random bit strings which generate samples are of the form* $x^i(0/1)^j 01^k$, *where* $i, j, k \in \mathbb{Z}^*$, $i+j+k+1 = n$ *and* x *is the* don't care *bits.*

Experimentally, we have seen that j is bounded by a *small* Δ i.e. $j_{\max} \leq \Delta$. For example for $\sigma = 1, 2, 6.15543$ and 215 we find the $\Delta = 4, 4, 6$, and 15, respectively. In what follows,

we show that these two facts allow to develop a very efficient minimization technique for a fast constant-time Gaussian sampler.

3.2.1 Efficient minimization

To instantiate an efficient discrete Gaussian sampler with a specific standard deviation, we first enumerate the number of leaves in the DDG tree and the random bit strings that hit those leaf nodes. We create a list L of these random bit strings $x^i(0/1)^j01^k$ with their corresponding binary decomposed sample values. It is evident from Sec. 3.1.2 that the size of this list is $\sum_{i=0}^{n-1} \mathrm{h_i}$.

Now, using this list L, it is possible to generate the Boolean functions f^ι that simply map the input random bit strings to the sample bits. To improve efficiency, we can use synthesis tools to minimize these Boolean expressions, since minimizing these Boolean expressions is equivalent to the well known circuit minimization problem which is a NP-complete problem. Also, as the number of variable n is large, these tools can only use heuristic minimization algorithms which can generate a number of complications. Firstly, the behaviour of these heuristic algorithms is very unpredictable which may cause the resulting minimized expressions or in turn the discrete Gaussian sampler to behave in a very unexpected manner. Secondly, minimization is not very efficient as the tools can use only heuristic algorithms. And lastly, most of these efficient heuristic algorithms are intellectual property of their respective corporations and the output of these algorithms cannot be put in the public domain. This renders them unusable for discrete Gaussian samplers for lattice based cryptography implementations which are mostly open source.

To overcome these problems we propose an alternative but efficient minimization strategy. We first sort the input random bit strings $x^i(0/1)^j01^k$ and their corresponding sample bits in the list L in the ascending order of k. This makes all the input random bit strings with equal number of consecutive 1's from the LSB become adjacent in the list L. This is shown in Fig. 4 for a discrete Gaussian sampler with $\sigma = 2$.

In the next step, we divide the list L in sublists $\mathrm{l}_0, \mathrm{l}_1, \cdots \mathrm{l}_{\mathrm{n}'}$ such that all the input random bit strings in the sublist l_κ have $\kappa \in [0, n']$ consecutive 1's from the LSB or are of the form $x^i(0/1)^j01^\kappa$. As in the sublist l_κ the $\kappa + 1$ least significant bits are fixed, the output sample bits in this sublist are determined by the next j random bits only. Now, we recall that $j_{\max} \leq \Delta$. Hence, we can generate Boolean functions f^ι_Δ that map Δ input random bits to the output sample bits for each sublist. Since Δ is *small*, we can now use very efficient minimization techniques to minimize the Boolean expressions f^ι_Δ. It is even practically feasible to use Karnaugh map or brute force techniques to minimize the expressions. In this work we used the open source tool `Espresso` with `-Dso -S1` options for *exact* minimization of each expression. We generate these Boolean expressions $f^{\iota,t}_\Delta, t \in [0, n']$ for all the sublists $\mathrm{l}_0, \mathrm{l}_1, \cdots \mathrm{l}_{\mathrm{n}'}$ and for all $\iota \in [0, m-1]$. In the next section, we will discuss how we can join these Boolean expressions together to create a constant-time discrete Gaussian sampler.

3.2.2 Constant-time sampling

We first recall a method to execute a non constant-time *if-else* block $\nu = \alpha \ ? \ \beta_0 : \beta_1$ in constant-time as $\nu = (\alpha \& \beta_0) | (\bar{\alpha} \& \beta_1)$ where α, β_i are binary variables. It is easy to extend this to a long *if-elseif-...-else* block as $\nu = (\alpha_0 \& \beta_0) | (\bar{\alpha}_0 \& ((\alpha_1 \& \beta_1) | (\bar{\alpha}_1 \& (\ldots | (\bar{\alpha}_n \& \beta_{n+1})))))$. Such methods for constant-time execution of *if-else* blocks have been known since as early as constant-time bit sliced implementation of DES [10] and their constant-time behaviour have been studied well in the literature.

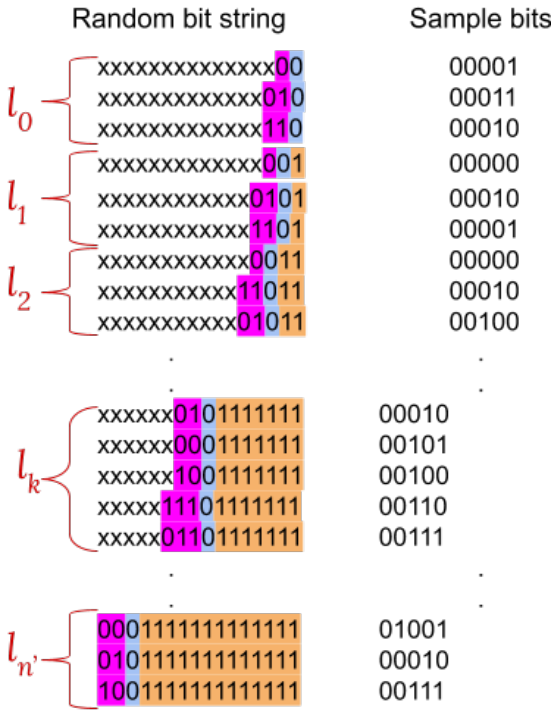

Figure 4: Dividing a List L in sublists l_κ for $n = 16$ and $\sigma = 2$. The rightmost bit is the LSB in both the columns.

Now, consider the binary variable $c_\kappa = b_0 \& b_1 \& \ldots \& b_{\kappa-1} \& \bar{b}_\kappa$ where b_i's are the input random bits to the Gaussian sampler. Also, we recall from the previous section that the input random bits of sublist l_κ are of the form $x^i(0/1)^j 01^\kappa$. Now, we make the following claim:

Claim 1. *c_κ equals 1 if and only if the random bit string belongs to sublist l_κ.*

The above claim can be proven easily by using the structure of input random bits of list l_κ and the definition of the variable c_κ. Using Claim 1 and the constant-time *if-elseif-...-else* blocks as discussed in the beginning of this section we can combine the Boolean expressions $f_\Delta^{\iota,t}, t \in [0, n']$ to create a constant time Gaussian sampler with precision n as shown in Eqn. 2,

$$f_n^\iota = c_0 \ ? \ f_\Delta^{\iota,0} : (c_1 \ ? \ f_\Delta^{\iota,1} : (\ldots : (c_{n'-1} \ ? \ f_\Delta^{\iota,n'-1} : f_\Delta^{\iota,n'}))) \tag{2}$$

So far, we used only binary variables $\alpha, \beta, c_\kappa, b$ etc., to describe our methods. All of these methods can be trivially transformed into the bit-sliced SIMD setting as described in [25] by using variables of larger bit lengths instead of binary variables, and by replacing single bit Boolean operators to their bit wise counterparts.

We assert that our construction of the Boolean expression f_n^ι runs in constant-time as long as each of the Boolean expressions $f_\Delta^{\iota,t}$ runs in constant-time. Now, each of $f_\Delta^{\iota,t}$ is a Boolean expression that computes ι-th bit of a sample from Δ random bits. The constant-time behaviour of such functions has been proven and analyzed rigorously in [25]. The whole process for efficient

minimization of f_n^ι is shown as a flowchart in Fig. 5. We provide a tool that implements the strategies mentioned here.[1]

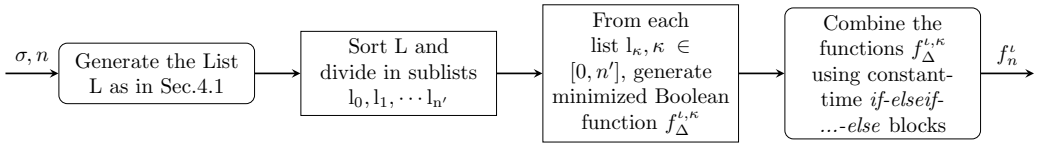

Figure 5: Flowchart efficient minimization of Boolean expressions f^ι for constant-time discrete Gaussian sampling.

Fig. 6 shows histogram plots of constant-time discrete Gaussian sampling for $\sigma = 2$ and 6.15543 using the methods described above.

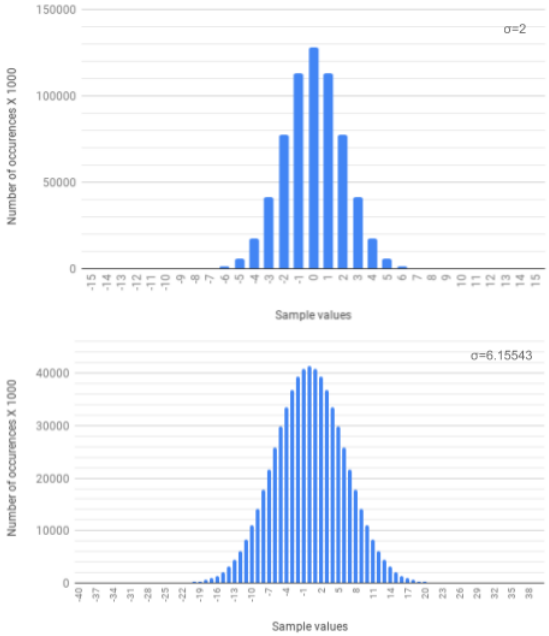

Figure 6: Histogram plot for $\sigma = 2$ and $\sigma = 6.15543$ using 64×10^7 samples.

	Constant-time sampler in [25]	This work	Improvement
$\sigma = 2$	3,787	2,293	37%
$\sigma = 6.15543$	11,136	9,880	11%

Table 1: Comparing discrete Gaussian sampler with our efficient minimization with the method previously described in [25]. The numbers in the table are in clockcycles and do not include the overhead for generating the pseudorandom numbers.

3.2.3 Performance evaluation

In Table 1, we compare the performance of the Gaussian sampler with our efficient minimization technique and the Gaussian sampler with simple minimization described in [25]. The computa-

[1] Tool and the code at https://github.com/Angshumank/const_gauss_split

tion times are measured on a single core of a Intel(R) Core(TM) i7-6600U processor running at 2.60GHz and disabling hyper-threading, Turbo-Boost, and multi-core support as standard practice on Ubuntu 16.04 running on a Dell Latitude E7470 laptop. As our target processor has 64-bits our sampler can generate 64 samples in a batch. We can see that for $\sigma = 2$ we get around 37% improvement, but the improvement for $\sigma = 6.15543$ is approximately 11%. The reason behind this is that in [25] the output of the minimization tool for $\sigma = 6.15543$ has been manually optimized further for efficiency.

4 Constant-time polynomial multiplication

Polynomial arithmetic is a computationally critical part in implementations of lattice-based cryptosystems. Polynomial addition and subtraction can easily be performed by coefficient-wise additions or subtractions in the underlying field, thus with $O(n)$ complexity. Here $n-1$ denotes the degree of the polynomial. This is however not the case for polynomial multiplication. In the general setting, the calculation of $c(x) = a(x) \cdot b(x) \bmod f(x)$ is done in two steps: first, by computing the product $c'(x) = a(x) \cdot b(x)$, and second, by reducing the result $c(x) = c'(x) \bmod f(x)$ modulo the irreducible polynomial $f(x)$.

Polynomial multiplication based on the Number Theoretic Transform (NTT) is believed to be the most efficient multiplication method due to its $O(n \log n)$ complexity and hence, parameters in most lattice-based schemes are chosen so that NTT can be used. In this Chapter we prove that a combination of Toom-Cook, Karatsuba and low-degree schoolbook multiplications can outperform optimized NTT-based polynomial multiplications in a microcontroller setting. Since polynomials are stored in a sequential addressable memory (typically SRAM), the memory access overhead plays a critical role in the overall performance that is not captured in the asymptotic notion of the algorithmic complexity. We propose optimization techniques that carefully use the ALU-registers to accommodate several coefficients of the polynomials at once and then compute as many operations as possible involving the *local* coefficients, thus effectively reducing the number of sequential memory accesses.

Our proposed technique is very modular and can be adapted to embedded platforms with different capabilities and/or constraints. We showcase this aspect by selecting as target platforms two ARM microcontrollers: the low-cost Cortex-M0, and the more powerful Cortex-M4. Both of them are popular for realizing Internet of Things (IoT) applications, and due to their role as edge platforms they represent typical targets for physical attacks. Our implementations have neither data dependent branches nor data-dependent table lookups. Thus they provide inherent protection against timing-based side-channel attacks. In addition, their performance makes them suitable as a starting point to build countermeasures against power/EM side-channel attacks, which typically incur large performance overheads. This aspect will be investigated in the upcoming months of the project.

The contents of this Chapter are extracted from the following FENTEC publications:

A. Karmakar, J. M. Bermudo Mera, S. Sinha Roy, I. Verbauwhede: *Saber on ARM CCA-secure module lattice-based key encapsulation on ARM*. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2018(3): 243-266 (2018)

J. M. Bermudo Mera, F. Turan, A. Karmakar, S. Sinha Roy, I. Verbauwhede: *Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism*. IACR Cryptology ePrint Archive. 2020: 321 (2020)

In what follows, we provide a high-level description of our speed-optimized and memory-optimized implementations for polynomial multiplication, suitable for resource-constrained edge platforms, as well as our hardware co-processor for accelerating polynomial multiplication. We showcase the efficiency of our techniques by instantiating them for the lattice-based Key-Encapsulation Mechanism (KEM) Saber [16], noting that they can be leveraged as building blocks for FE implementations based on lattice problems.

For detailed technical information, we refer the reader to the complete works in [24] and in [36].

4.1 Preliminaries

In this section, we first give a quick description of the Saber KEM and detail its parameter set. We provide minimal information necessary to understand how our method can be instantiated. For more details on the mathematical structure of Saber, algorithms and security proofs, we refer the reader to [16]. Next, we describe the baseline techniques for polynomial multiplication that are at the core of our implementations.

4.1.1 The Saber KEM

The security of Saber relies on the module-LWR problem, which is similar to the module-LWE problem but uses rounding to introduce the errors. During key generation, a 32 bytes random seed $seed_{\boldsymbol{A}}$ is generated and later expanded to construct the pseudorandom public matrix $\boldsymbol{A}$ of dimension $l \times l$. The secret $\boldsymbol{s}$ is a vector of dimension l and is sampled from a centered binomial distribution β_μ with parameter μ. A vector $\boldsymbol{b}$ is computed by performing a matrix-vector multiplication of $\boldsymbol{A}$ and $\boldsymbol{s}$ followed by an addition of a constant vector $\boldsymbol{h}$ and then bit-selection (using the `bits`) from the result. The public-key consists of $\boldsymbol{b}$ and the seed. The encryption operation uses matrix generation, binomial sampling, matrix-vector multiplication and bit selection. The decryption operation is rather simple and uses vector-vector multiplication and bit selection.

For around 180-bit of quantum-security, Saber uses a matrix or vector dimension $l = 3$ and ring-dimension $n = 256$. The two moduli p and q are 2^{10} and 2^{13} respectively. The parameter of the binomial error distribution is $\mu = 8$.

Saber performs polynomial arithmetic in the ring $R_q = \mathbb{Z}_q[x]/\langle x^{256} + 1\rangle$. The matrix-vector multiplications are very costly and therefore need special care. But since the parameter q is not a prime number, implementations of Saber cannot take advantage of the NTT-based polynomial multiplication. This aspect motivates the polynomial multiplication approach presented in this chapter, which uses a combination of methods due to Karatsuba and Toom-Cook.

4.1.2 Polynomial multiplication

Karatsuba Method. The Karatsuba algorithm [23] uses a divide-and-conquer approach to achieve $O(n^{\log_2 3})$ time complexity. The input polynomials $f(x)$ and $g(x)$ are split into half-sized polynomials as $f(x) = f_0 + f_1 \cdot x^{n/2}$ and $g(x) = g_0 + g_1 \cdot x^{n/2}$ and then the product is computed as $f(x) * g(x) = f_0 \cdot g_0 + \big((f_0 + f_1) \cdot (g_0 + g_1) - f_0 \cdot g_0 - f_1 \cdot g_1\big) \cdot x^{n/2} + f_1 \cdot g_1 \cdot x^n$. The algorithm is applied recursively. We refer to this algorithm as the classical Karatsuba in the rest of this chapter.

Memory-efficient Karatsuba Method. The classical Karatsuba algorithm requires an additional $O(n)$ memory at each recursive call. Thus it is rather "memory hungry" for multiplying large polynomials on resource constrained microcontrollers such as Cortex-M0. In 2009, Roche [44] proposed a modification of Karatsuba's algorithm that requires only $O(\log n)$ additional memory per recursion. For the input polynomials $f^{(0)}(x)$, $f^{(1)}(x)$ and $g(x)$ the algorithm computes $h(x) = h(x) + \big(f^{(0)}(x) + f^{(1)}(x)\big) \cdot g(x)$. It works by performing rearrangements of the output array between each of the three recursive calls so that all write back operations can be performed only on the output array of length $2n - 1$, as shown in Algorithm 2. The objective

Algorithm 2: Memory efficient Karatsuba `kara_mem` [44]

Input: Three polynomials $f^{(0)}(x)$, $f^{(1)}(x)$ and $g(x)$ and their degree n
Output: $h(x) = h(x) + \left(f^{(0)}(x) + f^{(1)}(x)\right) \cdot g(x)$ of degree $2n-1$

1 $k = n/2$
2 $h[k : 2k-1] = h[0 : k-1] + h[k : 2k-1]$
3 $h[3k-1 : 4k-2] = f^{(0)}[0 : k-1] + f^{(1)}[0 : k-1] + f^{(0)}[k : 2k-1] + f^{(1)}[k : 2k-1]$
4 **recursive call**: $h[k :] =$`kara_mem`$(g[0 :], g[k :], h[3k-1 :], k)$;
5 $h[3k-1 : 4k-2] = h[k : 2k-1] + h[2k : 3k-2]$
6 **recursive call**: $h[0 :] =$`kara_mem`$(f^{(0)}[0 :], f^{(1)}[0 :], g[0 :], k)$;
7 $h[2k : 2k-2] = h[2k : 3k-2] - h[k : 2k-2]$
8 $h[k : 2k-1] = h[3k-1 : 4k-2] - h[0 : k-1]$
9 **recursive call**: $h[2k :] =$`kara_mem`$(f^{(0)}[k :], f^{(1)}[k :], g[k :], k)$;
10 $h[k : 2k-1] = h[k : 2k-1] - h[2k : 3k-1]$
11 $h[2k : 3k-2] = h[2k : 3k-2] - h[3k : 4k-2]$
12 **return** $h(x)$;

of these rearrangements is to prepare the arrays for the next recursive call as well as to cancel the terms that were added in the previous calls as a consequence of the multiply-and-accumulate nature of this algorithm.

Toom-Cook method. The Toom-Cook method [28] is a generalization of the Karatsuba method: each multiplicand polynomial is split into w polynomials each having n/w coefficients. This is known as w-way Toom-Cook multiplication. A Toom-Cook multiplication essentially consists of three main steps: splitting, evaluation, and interpolation. In [16], the authors of Saber used a four-way Toom-Cook multiplication (see Alg. 3) to split a 256 multiplication into seven 64×64 multiplications. In the splitting stage, each multiplicand polynomial A and B is split into four equal polynomials each having 64 coefficients as $A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0 \cdot y$, (and similarly for B) where $y = x^{64}$. In the evaluation phase, these polynomials are evaluated at the points $\{0, \pm 1, \pm\frac{1}{2}, 2, \infty\}$ resulting in weighted sums of polynomials A_i's and B_i's. These weighted sums are then multiplied with each other to create polynomials w_1 to w_7 (steps $3-9$ in Alg. 3). In the interpolation stage (steps $10-24$) these values are further processed to produce the result C.

Algorithm 3: Toom-Cook Algorithm [16]

Input: Two polynomials $A(x)$ and $B(x)$ of degree $n = 256$
Output: $C(x) = A(x) * b(x)$
// Splitting $A(x)$ into four polynomials of size 64
$A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{64}$
// Splitting $B(x)$ into four polynomials of size 64
$B(y) = B_3 \cdot y^3 + B_2 \cdot y^2 + B_1 \cdot y + B_0$
// Evaluation of the polynomials at $y = \{0, \pm 1, \pm\frac{1}{2}, 2, \infty\}$.
$w_1 = A(\infty) * B(\infty) = A_3 * B_3$
$w_2 = A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3)$
$w_3 = A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3)$
$w_4 = A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3)$
$w_5 = A(\frac{1}{2}) * B(\frac{1}{2}) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3)$
$w_6 = A(\frac{-1}{2}) * B(\frac{-1}{2}) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3)$
$w_7 = A(0) * B(0) = A_0 * B_0$
// Interpolation
$w_2 = w_2 + w_5$
$w_6 = w_6 - w_5$
$w_4 = (w_4 - w_3)/2$
$w_5 = w_5 - w_1 - 64 \cdot w_7$
$w_3 = w_3 + w_4$
$w_5 = 2 \cdot w_5 + w_6$
$w_2 = w_2 - 65 \cdot w_3$
$w_3 = w_3 - w_7 - w_1$
$w_2 = w_2 + 45 \cdot w_3$
$w_5 = (w_5 - 8 \cdot w_3)/24$
$w_6 = w_6 + w_2$
$w_2 = (w_2 + 16 \cdot w_4)/18$
$w_3 = w_3 - w_5$
$w_4 = -(w_4 + w_2)$
$w_6 = (30 \cdot w_2 - w_6)/60$
$w_2 = w_2 - w_6$
return $C(y) = w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7$;

4.2 Our fast method

In this section we detail how to implement our polynomial multiplication method in the embedded platform ARM Cortex-M4 for Saber. The optimization target is speed. To this end, we leverage on the availability of a Digital Processing Unit (DSP) in the Cortex-M4 that allows for efficient multiply-accumulate instructions. The complete work in [24] describes an alternative implementation for ARM Cortex-M0. In this case the optimization target is memory footprint, given that the Cortex-M0 is designed for low-power and features limited memory resources. In this deliverable we skip the description of this implementation, and focus instead on the one on ARM Cortex-M4.

The polynomials from Saber are stored in the memory of the microcontroller as arrays of 16-bit

half-words, each half-word containing a coefficient of the polynomial. For fast computation of polynomial multiplication, we first apply the Toom-Cook method to split a 256-coefficient polynomial multiplication into seven 64×64 polynomial multiplications. Then each such polynomial multiplication is performed using the Karatsuba method [23] until the number of coefficients in the operand polynomials becomes 16. Each 16-coefficient polynomial multiplication is computed using the quadratic-complexity schoolbook method. Since the coefficients are stored in the sequential addressable memory which is SRAM in our implementation, the overhead of memory accesses plays a critical role in the overall performance, and is not counted when considering the notion of asymptotic complexity of the polynomial multiplication algorithms. In what follows, we show how we minimize the memory access overhead by designing efficient algorithms.

4.2.1 Memory access optimization in Toom-Cook evaluation and interpolation

In the evaluation phase of the four-way Toom-Cook algorithm (Alg. 3), weighted sums of the polynomials A_0 to A_3 and B_0 to B_3 are computed to construct the multiplicands of the next-level polynomial multiplications. If the evaluations are computed *horizontally*, i.e., compute the polynomials $A(y)$ and $B(y)$ for the particular evaluation point and then compute $A(y) * B(y)$, then the cost of memory access will be huge. This is because for each weighted sum we have to read all the coefficients of A_0 to A_3 and B_0 to B_3 from the memory.

To minimize the number of memory accesses, we adapt a *vertical* coefficient scanning method. The jth coefficients of the four polynomials A_0 to A_3 are read in a batch from the memory and loaded in the registers of the processor. Then these registers are used to compute the jth coefficients of all of the weighted sums as required in the lines 4 to 8 in Alg. 3. We can do this easily as there are 14 usable general purpose registers in Cortex-M4 and this approach needs only 8 registers as shown in Alg. 4. In the algorithm, r_* represent the general purpose registers and $aws_*[\,]$ represent the arrays of the weighted sums that are stored in the memory. In Alg. 4, special care has been taken to minimize the number of arithmetic operations during this process. We repeat the same procedure to compute the weighted-sum arrays from B_0 to B_3. Unlike the horizontal method, the vertical method accesses each of the 64 coefficients of A_0 to A_3 and B_0 to B_3 only once. The overhead of this approach is that it requires ten additional arrays each of length 64 to store the weighted sums.

During the interpolation phase of the Toom-Cook algorithm, we apply a similar vertical technique: we load the jth coefficients of all of w_1 to w_7 in the internal registers and perform the arithmetic operations on the registers. Having multiple consecutive loads also helps to decrease the latency since atomic load instructions take three clock cycles, whereas batch loading of three coefficients takes only four cycles thanks to the pipelined datapath of Cortex-M4. The optimized steps are shown in Alg. 5.

Algorithm 4: Toom-Cook 4-way evaluation

Input: Two polynomials $A(x)$ and $B(x)$of degree $n = 256$
Output: Evaluation polynomials w_1 to w_7 as in Alg. 3

for $j = 0$ *to* 63 **do**
- $r_0 = A_0[j]$;
- $r_1 = A_1[j]$;
- $r_2 = A_2[j]$;
- $r_3 = A_3[j]$;
- $r_4 = r_0 + r_2$;// $A_0 + A_2$
- $r_5 = r_1 + r_3$;// $A_1 + A_3$
- $r_6 = r_4 + r_5$; $r_7 = r_4 - r_5$;
- $aws_3[j] = r_6$;
- $aws_4[j] = r_7$;
- $r_4 = 2*(r_0*4+r_2)$;// $8*A_0+2*A_2$
- $r_5 = r_1*4+r_3$;// $4*A_1+A_3$
- $r_6 = r_4 + r_5$; $r_7 = r_4 - r_5$;
- $aws_5[j] = r_6$;
- $aws_6[j] = r_7$;
- $r_4 = 8*r_3 + 4*r_2 + 2*r_1 + r_0$;
- $aws_2[j] = r_4$; $aws_7[j] = r_0$;
- $aws_1[j] = r_3$;

Repeat the above steps to generate weighted sums bws_1 to bws_7;

for $i = 1$ *to* 7 **do**
- $w_i = aws_i * bws_i$;

return w_1 *to* w_7;

Algorithm 5: Toom-Cook 4-way interpolation

Input: Evaluation polynomials w_1 to w_7 as in Alg. 3
Output: $C(x) = A(x) * B(x)$ as in Alg. 3

$C \leftarrow 0$;
for $j = 0$ *to* 126 **do**
- $r_1 = w_2[i]$; $r_4 = w_5[i]$;
- $r_5 = w_6[i]$; $r_0 = w_1[i]$;
- $r_2 = w_3[i]$; $r_3 = w_4[i]$;
- $r_6 = w_7[i]$;
- $r_1 = r_1 + r_4$; $r_5 = r_5 - r_4$
- $r_3 = (r_3 - r_2)/2$; $r_4 = r_4 - r_0$
- $r_8 = 64 \cdot r_6$; $r_4 = r_4 - r_8$
- $r_4 = 2 \cdot r_4 + r_5$; $r_2 = r_2 + r_3$
- $r_1 = r_1 - 65 \cdot r_2$; $r_2 = r_2 - r_6$
- $r_2 = r_2 - r_0$; $r_1 = r_1 + 45 \cdot r_2$
- $r_4 = (r_4 - 8 \cdot r_2)/_{24}$; $r_5 = r_5 + r_1$
- $r_1 = (r_1 + 16 \cdot r_3)/_{18}$; $r_3 = -(r_3 + r_1)$
- $r_5 = (30 \cdot r_1 - r_5)/_{60}$; $r_2 = r_2 - r_4$
- $r_1 = r_1 - r_5$; $C[i] = (C[i] + r_6)$;
- $C[64 + i] = (C[64 + i] + r_5)$;
- $C[128 + i] = (C[128 + i] + r_4)$;
- $C[192 + i] = (C[192 + i] + r_3)$;
- $C[256 + i] = (C[256 + i] + r_2)$;
- $C[320 + i] = (C[320 + i] + r_1)$;
- $C[384 + i] = (C[384 + i] + r_0)$;

return C;

To validate that this algorithm is more efficient, we have implemented both algorithms on C prior to realizing any assembly optimization on the Toom-Cook function. In this setting, the *horizontal* algorithm executes a 256 coefficient multiplication in $83,550$ clock cycles while the *vertical* algorithm only requires $78,633$ clock cycles for the same multiplication, using in both algorithms the same combination of two level Karatsuba and schoolbook behind Toom-Cook. Then, we have also carried out assembly optimizations to achieve our cycle counts as shown in Table 2.

4.2.2 Speeding up School book multiplication using DSP instructions

As Saber uses LWR, it needs multiplication for two different rings. Hence, the coefficients of each polynomial can be either 10 or 13 bits long. We pack each coefficient of a polynomial in a 16-bit half-word. Two coefficients are then loaded in a single 32-bit full-word of the processor. Let r^i be a full-word register. We represent the bottom half-word by r^i_0 and the top half-word by r^i_1. We use the DSP multiply-and-accumulate instruction `SMLA` available in Cortex-M4 for multiplying two half-words and accumulating the result. The flags B or T in the instruction are used for choosing the bottom or top half-word respectively.

$$\texttt{SMLA}^{B/T B/T}(r^a, r^b, r^c, r^d) := r^a \leftarrow r^b_{0/1} * r^c_{0/1} \ + \ r^d$$

We use this instruction to multiply two coefficients or to compute the halfword multiplications like a_0b_0, a_2b_0 as shown in Fig. 7.

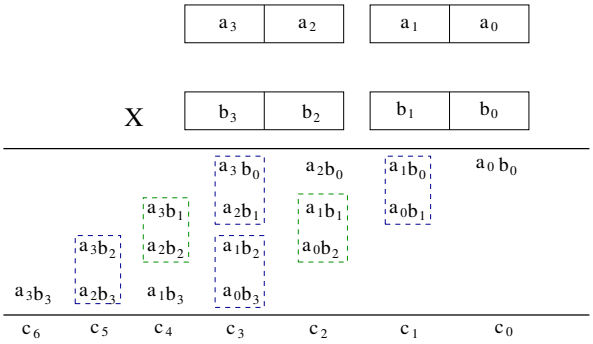

Figure 7: Reducing the number of multiply-and-accumulate instructions in schoolbook multiplication.

Now, consider again the example shown in Fig. 7 where we multiply four coefficients of polynomial $A(x)$ and four coefficients of polynomial $B(x)$ and accumulate the result in the polynomial $C(x)$. In the naive way, we have to use 16 `SMLA` instructions in this computation. We use the DSP instruction `SMLADX` that facilitates cross multiplication between half-words of registers and perform two multiply-and-accumulate operations (marked in blue dashed rectangles) simultaneously.

$$\texttt{SMLADX}(r^a, r^b, r^c, r^d) := r^a \leftarrow r_0^b * r_1^c \ + \ r_1^b * r_0^c \ + \ r^d$$

This effectively reduces the total number of multiplication instructions in Fig. 7 from 16 to 12. For multiplying two polynomials with 16 coefficients we need only 192 multiply-and-accumulate operations instead of 256. Further, with some arrangements of the coefficients we can do even better. In the beginning of each inner loop of a multiplication, we pack two adjacent coefficients that are in different registers (e.g., coefficients b_1 and b_2) in a spare register using `PKHBT` instruction and then perform cross multiplication using `SMLADX` instruction as explained above(marked in green dashed rectangles). In our assembly routine for schoolbook multiplication, we are able to fit a maximum of eight coefficients of one multiplicand polynomial and four coefficients of the other multiplicand polynomial at a time due to our optimized usage of internal registers. As the input polynomials to our schoolbook multiplication routine are always of 16 coefficients, the inner loop of our schoolbook multiplication runs only eight times. Thus in each inner loop we can save four instructions due to the rearrangement of coefficients at the cost of one extra `PKHBT` for the rearrangement. Hence, we can save three instructions per iteration of the inner loop. So, ultimately we need only 168 multiply-and-accumulate instructions instead of 256 instructions resulting in approximately 34% less multiply-and-accumulate instructions. Our assembly-optimized schoolbook multiplication takes only 587 clock cycles for a 16×16 polynomial multiplication. It should be noted that this method can not be used in microcontrollers without similar DSP instructions, for example ARM Cortex-M0.

4.3 Results

We use the ARM-GCC toolchain to compile our source code with the flags `-O3` for the ARM Cortex-M4. The clock-cycles are measured with inbuilt functions using a clock running at the same frequency as the processor. We executed the software 10,000 times and found that each execution requires the same number of cycles thus supporting our claim that our implementation is indeed constant-time. We used an STM32F4-discovery board from STMicroelectronics to evaluate the performance on a Cortex-M4 platform. This platform is equipped with a random

number generator which we also use in our implementation to generate the seed bytes as specified in the Saber scheme.

Given that our implementation is highly modular in nature, it supports different trade-offs between speed and memory usage. Fig. 8 shows variations in time and memory for different optimizations in the Cortex-M4 implementation. In the figure, the abbreviation *TC* refers to Toom-Cook, *kara_mem* to memory efficient Karatsuba and *kara_classic* to the classical Karatsuba algorithm, respectively.

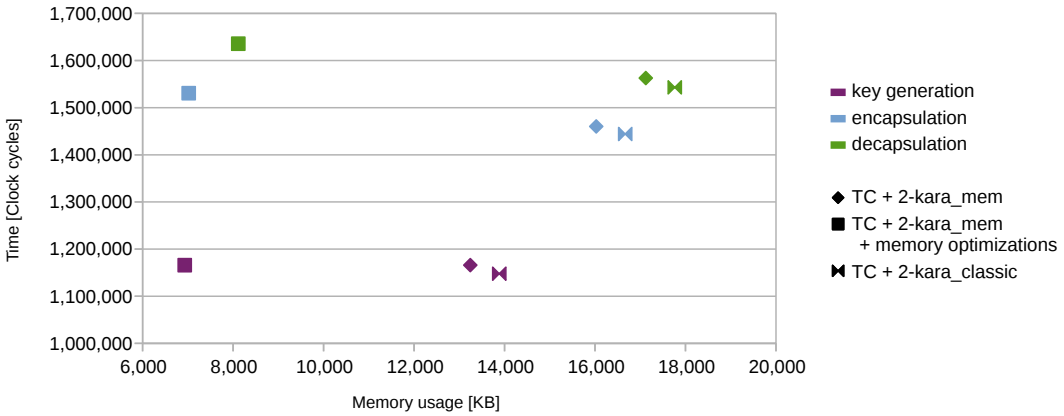

Figure 8: Time vs memory for different combinations of optimizations in Cortex-M4.

Table 2 shows the number of clock cycles required to perform a single 256×256 polynomial multiplication. Since [37, 5] report results for NTT-based multiplications of 512 and 1024-coefficient polynomials, we scaled down the cycle counts by factor 2.25 and 5 respectively considering the asymptotic $O(n \log n)$ complexity of the NTT. Here we can see that our fastest multiplication outperforms the NTT-based multiplications in [37, 17, 41].

To the best of our knowledge we gain this advantage over NTT-based multiplications firstly because Saber uses a power-of-two modulus and thus we do not spend any time for modular reduction. Secondly, NTT has a complex butterfly structure which requires access to non-consecutive memory words. In contrast, our combination of Toom-Cook, Karatsuba and schoolbook methods accesses memory in a consecutive fashion and hence we can use the DSP instructions in a better way than the NTT based multiplications. These expensive memory access and modular reduction operations counteract the asymptotical advantage of the NTT-based polynomial multiplication.

4.4 HW-SW codesign of Saber

In this section we show how to take advantage of heterogeneous platforms to accelerate an already optimized software implementation by means of hardware-software co-design techniques. The domain-specific accelerator is designed following a hardware-software co-design approach in order to take advantage of the custom logic that can be implemented in an FPGA to accelerate the most computationally expensive operations while maintaining the flexibility offered by a micro-controller to control the execution flow. As the most expensive operation in Saber is the polynomial multiplication, which has to be executed l^2 times during matrix-vector multiplication, its computation is offloaded to hardware.

Implementations	Polynomial multiplication
Cortex-M4F [37]†	226,055
Cortex-M4F [17]‡	108,147
Cortex-M4 [41, 12] ▷	≈73,705
Cortex-M4 [5]*	≈54,447
Ours ⊕	65,459
Ours ⋆	66,692

† Reported 508,624 cycles for polynomial degree 512 and prime modulus 12289.

‡ Reported 108,147 cycles for polynomial degree 256 and prime modulus 7681.

∗ Reported ≈ 272,235 cycles for polynomial degree 1024 and prime modulus 12289.

▷ Polynomial degree 256 and prime modulus 7681

⊕ Speed-optimized implementation. Toom-Cook+classical Karatsuba+schoolbook

⋆ Speed-optimized implementation. Toom-Cook+memory-efficient Karatsuba+schoolbook

Table 2: Comparison of clock cycles for 256×256 polynomial multiplications with scaling when necessary. The cycle counts for NTT based multiplications are calculated as cycle count for 2*Forward NTT + Inverse NTT.

4.4.1 Architecture

When setting the boundaries between the operations that will be offloaded to hardware and those executed in the micro-controller, we exploit parallelism at system level by generating the polynomials needed for the next multiplication in software while the hardware performs the arithmetic on the previous operands. This approach pipelines the generation of the polynomials with the arithmetic operations, improving the performance as well as the utilization of the available resources.

We implemented our co-processor on a Xilinx Zynq device which integrates FPGA to ARM processors. Zynq devices support an AXI based communication interface for the interaction of ARM cores and any hardware module in FPGA. Additionally, Xilinx DMA offers the highest performance for bulky data transfers between memory and the modules. Hence, we tailored our interfacing mechanisms for an efficient use. We kept the data word size 64-bit for handling the coefficients both in the ARM side software and in the BRAM. As a result, the software stores the polynomials as arrays of 64-bit data words. The BRAM also uses the same data word length, hence we configured the DMA for transferring polynomial arrays from memory to BRAM directly. When performing these transfers, the DMA accesses the array with its memory address, and delivers it over (or receives from) the wrapper as a stream, i.e., one data word at each clock cycle. This stream is free from address information, hence our wrapper associates data words with an address in the BRAM. For associating the right address, the wrapper is informed with the transfer's base address by a command prior to the transfer's start. A register unit is used to support the interfacing with command and status registers. This approach makes the software side the master of our architecture, responsible of sending commands to the co-processor, observing its execution status and handling data transfers. Currently, commands for data transfers, evaluation, multiplication, MAC and interpolation are supported. The instruction-set architecture (ISA) is quite flexible leaving room for the inclusion of new commands or even the integration of more modules to accelerate other operations utilizing the same co-processor. Fig. 9 shows an overview of the system architecture.

Algorithmically, the polynomial multiplication is implemented using the four-way Toom-Cook

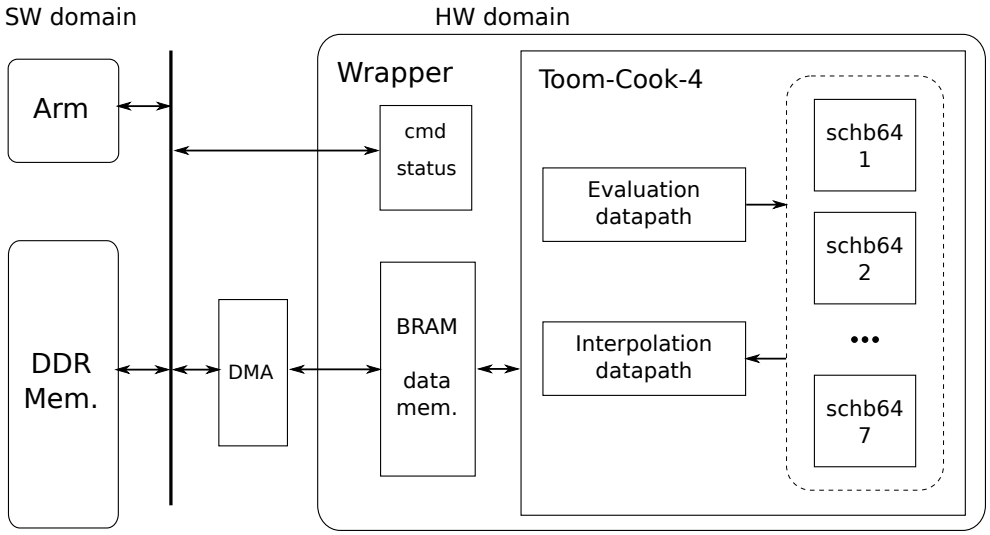

Figure 9: High-level architecture and interfacing of hardware and software.

with the vertical scanning described in Sec. 4.2.1 to break down a single multiplication of polynomials with 256 coefficients into 7 multiplications of polynomials with 64 coefficients that are performed in parallel by small multipliers. In the rest of this subsection we describe each of these components following a bottom-up approach.

4.4.2 64-coefficient polynomial multiplier

This unit is responsible for computing the 64-coefficients polynomial multiplications during Toom-Cook point-wise product. It will be instantiated seven times in parallel, so it is necessary to keep it simple to lower the area requirements of the overall design. For this reason, we choose straightforward schoolbook polynomial multiplication. We exploit parallelism once again and propose the generic architecture depicted in Fig. 10. First, there is a loading stage where $n_m = 4$ coefficients from b are loaded into the rightmost inputs of the multipliers, implemented with fabric DSPs. Then, all 64 coefficients in a are loaded consecutively into the other register. During this phase, one coefficient of the result is produced each clock cycle in the leftmost output register, while the rest of the accumulated intermediate values shift to the left. After this, n_m additional coefficients are produced while the datapath is flushed. Then, the next n_m coefficients in b are loaded and the process repeats until the full multiplication has been computed. The pipeline strategy is not trivial due to data dependencies between the accumulation and the previous result generated in the immediate right MAC. The critical path is shown in Fig. 10 with a red dashed line. To break it down without altering the dataflow, pipeline registers are included only in the multiplier. These pipeline registers are represented as green lines.

Since fabric LUTs have a depth of 64 bits, which matches the length of the polynomials, the distributed memory is implemented as LUT-based memory. Operand a is accessed sequentially, so a single port RAM is enough to store it. To halve the latency of the loading stage, operand b is stored in a dual port RAM. To allow simultaneous read and write operations, the result is stored in a dual port RAM. The data width is 16 bits because the modulus is $q = 2^13$ and Toom-Cook interpolation requires 3 extra bits of precision for the divisions by 8 during the interpolation step.

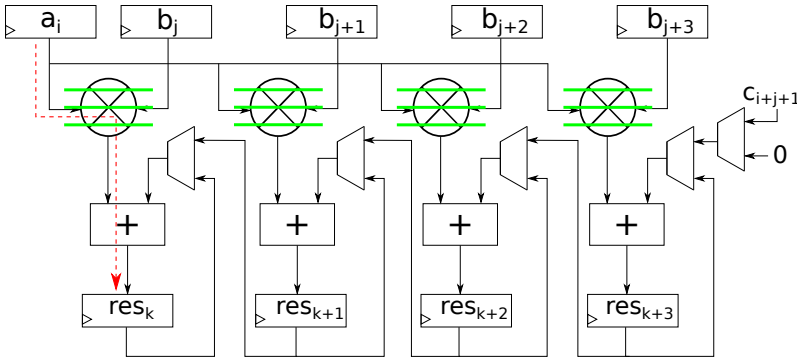

Figure 10: Architecture for the 64×64 polynomial multiplier utilizing 4 DSP units, including the critical path and the pipeline registers that break it down

To conclude the design-space exploration of this module, we show the impact of changing the number of DSPs, n_m. In our design, n_m affects the latency as in (3). The four terms in the equation correspond to (1) loading the n_m coefficients from b, (2) filling up the datapath, (3) performing the computation and (4) flushing the datapath between iterations. For a compact design we have chosen $n_m = 4$, i.e., 1168 clock cycles. It is the smallest option that allows our multiplier to be competitive with NTT, e.g., 1289 clock cycles [7]. For a high-performance implementation $n_m = 8, 16$ can be considered.

$$latency = \frac{64}{n_m}\left(\frac{n_m}{2} + 4 + 64 + (n_m - 1)\right) = \frac{4288}{n_m} + 96 \tag{3}$$

4.4.3 Toom-Cook multiplier

The three steps of Toom-Cook are implemented on different datapaths with independent control. The memory requirements of this module depend exclusively on how evaluation and interpolation are implemented. In particular, the reading throughput of the system memory imposes a performance limitation on the evaluation hardware while the interpolation hardware must accommodate to the writing pattern. Sec. 4.4.4 details the memory requirements, the access pattern, the address generation and the memory layout. In the following we focus on the evaluation and interpolation circuits.

Evaluation hardware

The evaluation datapath is derived from Alg. 3 as shown in Fig. 11. The same datapath is used to perform the evaluation for both operands, a and b, one immediately after the other. The weighted polynomials are directly stored in the distributed memory of the seven small polynomial multipliers. The delay introduced by two 16-bit adders is not big enough to require pipelining. The latency of the entire evaluation step is 128 clock cycles, which corresponds to reading the two 256-coefficient multiplicands in chunks of four coefficients per clock cycle.

Interpolation hardware

Building the hardware to execute the interpolation step is a challenging task because a direct mapping of Alg. 5 as for evaluation would result in a very asymmetric datapath with a long critical path. Instead, we identify certain symmetries in the interpolation matrix together with a trial and error approach to derive the circuit in Fig. 12. The critical path, indicated with a red dashed line, is broken down with pipeline registers, represented as horizontal green lines to allow a

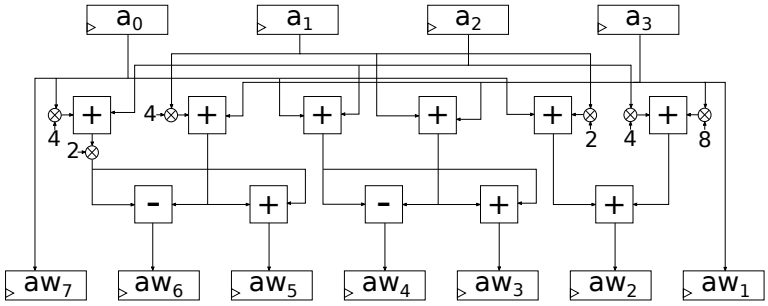

Figure 11: Datapath for the evaluation step

higher clock frequency. The interpolation hardware can read seven coefficients in parallel coming from the seven small polynomial multipliers but write operations can only be done at the clock rate due to the irregular memory accesses. Thus, memory operations become the bottleneck for interpolation. Irregular memory accesses are caused by the polynomial indexing. Interpolation outputs seven coefficients that must be written with offsets equal to $\{0, 64, 128, 192, 256, 320, 384\}$. The six least significant bits of the iteration counter can be used to generate the base address of the corresponding iteration while the offsets set the most significant bits of the writing address. However, the 127 iterations will end up mismatching the offsets and making inefficient a possible memory alignment to increase the writing throughput.

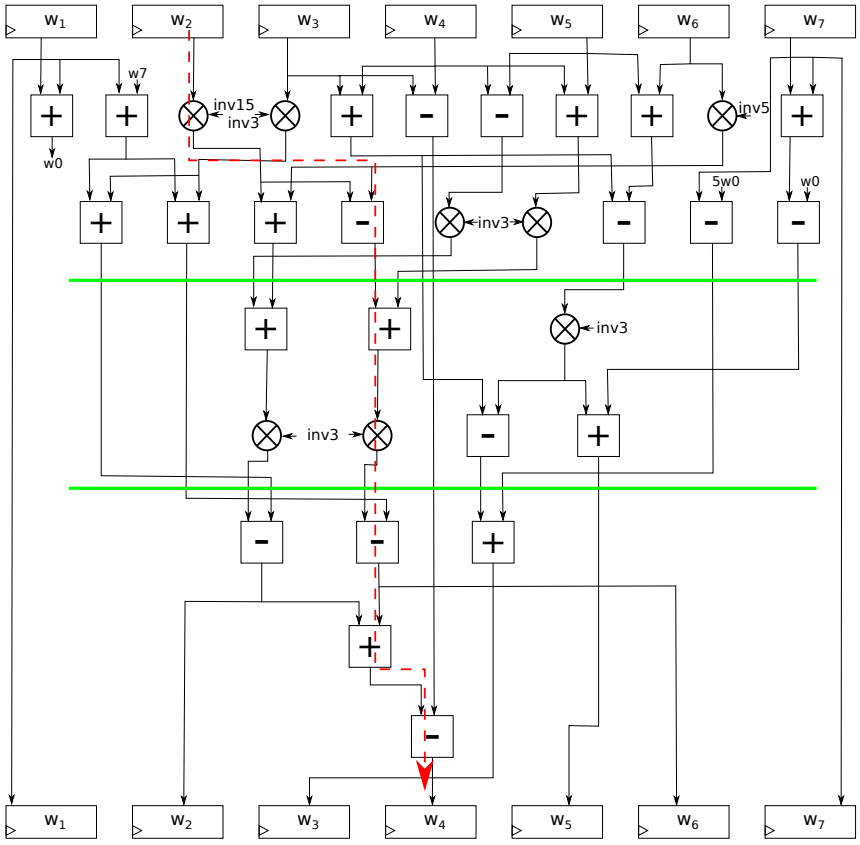

Figure 12: Datapath for the interpolation step including the critical path and the pipeline registers that break it down

4.4.4 On-chip memory

System memory is implemented using dedicated block RAM primitives called BRAM36K which can store up to 1024 words of 36 bits. For 64-bit words required by the evaluation stage, 2 BRAMs are needed. The memory is configured as dual port to allow simultaneous read and write operations, and with asymmetric read and write operations for the same port since read operations are performed on 64-bit words while write operations are performed on 16-bit words. One of the ports is also multiplexed between the HW/SW interfacing and the accelerator. Regarding the memory layout, the four coefficients that evaluation reads every clock cycle are not consecutive but offset with $0, 64, 128$ and 192. Then, coefficients must be aligned as in Fig. 13. Besides the coefficient alignment, polynomials are also aligned to 64 words, which is the natural alignment for 256-coefficient polynomials but not for 512-coefficient polynomials as shown in the same figure.

This memory layout can be created with almost no overhead when realizing the data transfer from software as it just requires a fixed offset on the indexing of the array. Since the memory is accessed asymmetrically for read and write operations, address translation is needed for computing the real writing address. When addressing the memory as a 16-bit word RAM, the least significant word of figure's address 0 corresponds to address 0, the next 16-bit word of figure's address 0 corresponds to address 1, etc. until the least significant 16-bit of figure's address 1, which corresponds to address 4 and so on. Since the memory is aligned to 256-coefficient polynomials, only the eight least significant bits of address need to be translated. Rewiring the two most significant bits from the coefficient index, which has a length of eight bits, to the two least significant bits of the address and shifting the other six two positions to the left gives the corresponding writing address for the coefficient.

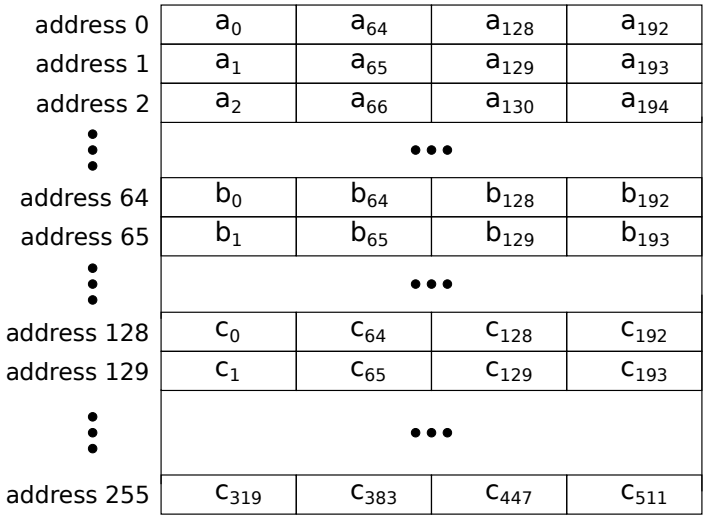

Figure 13: Coefficient alignment used in the system memory

4.4.5 Results

We implemented our domain-specific co-processor in the Xilinx ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. Software has been adapted from [24] by substituting their custom assembly optimizations by C code compiled with the GCC version available in the Xilinx SDK development tool. Hardware has been synthesized, placed and routed using Vivado 2018.1. Although

Table 3: Execution time (measured in million CPU cycles)

	only SW	SW/HW	Improvement
Key Generation	11.761	2.180	5.4
Encapsulation	14.944	2.762	5.4
Decapsulation	17.983	2.560	7.0
Polynomial Mult.	1.097	0.041	26.7

different synthesis and implementation strategies can be explored for a fine-grained optimized design, all results reported correspond to default configurations where the hardware co-processor runs at 125 MHz and the ARM processor runs at 666 MHz.

Performance results

Table 3 presents the performance of key generation, encapsulation, decapsulation and polynomial multiplication measured from software, columns show the execution time when using only software and the full co-processor. Saber becomes between 5.4 and 7 times faster while polynomial multiplication is almost 27 times faster. The execution time of the multiplication includes the overhead due to data transfers. In practice, many of these transfers are not necessary because we are performing matrix-vector multiplication instead of a standalone polynomial multiplication. Arithmetic operations only take 11835 clock cycles, which is 91 times faster than software even though the hardware is clocked more than five times slower than the software.

The overhead introduced by the commands sent from software to hardware is negligible due to the parallel transfer. However, this is not the case for the data transfers to the BRAM. Sending a polynomial, i.e., 512 bytes, from ARM to the co-processor takes 1816 clock cycles. Sending two polynomials, i.e., 1024 bytes, takes 2908 clock cycles. Larger data transfers are not of interest in our use case since polynomials are generated just-in-time on the CPU while the hardware runs the multiplication with the previous operands. Transfers in the other direction have almost the same execution times.

Resource utilization and comparisons with other works

The utilization of a single 64-coefficient polynomial multiplier including the LUT-based memory is of 342 LUTs, 155 FFs and 4 DSPs. This module is instantiated seven times and constitutes the core of arithmetic operations. The full hardware co-processor, including Toom-Cook multiplier, system memory and command decoding is implemented using 2927 LUTs, 1279 FFs, 2 BRAMs and 38 DSPs, which is quite a compact design.

Table 4 shows a comparison of our co-processor with other hardware implementations of NIST PQC second round candidates. For our work, we report the utilization of the full system including the processing system and the HW/SW interfacing. We can observe that module-LWR offers a trade-off between LWE, e.g., Frodo, and ring-LWE, e.g., NewHope. Comparison to an ASIC implementation is more difficult but Kyber is a scheme more similar to Saber. The implementation of [15] is a high-performance implementation of Saber on a superior FPGA technology.

Table 4: Comparison with state-of-the-art.

Scheme	Platform	Time [μs] KeyGen/Encaps/ Decaps	Freq [MHz]	BRAM/ DSP	FF/ LUT
Kyber [7]	ASIC	1548/2465/ 1646	72	-	-
Saber [15]	UltraScale+	- /60/ 65	322	4 / 256	11619/ 12566
Frodo [22]	Artix-7	45454/45454/ 47619	167	24 / 1	3559/ 7773
NewHope [31]†	Artix-7	51.9/78.6/ 21.1	133	14 / 8	9975/ 20826
Saber [36]	Artix-7	3273/4147/ 3844	125	2 / 28	7331/ 7400

†Implements only CPA secure NewHope

5 Functional encryption based on ring learning with errors

In this section, we describe our ring-learning with errors (RLWE) based functional encryption scheme. Earlier, functional encryption schemes based on standard learning with errors have been designed [1]. However, the main motivation behind designing a RLWE-FE is the huge computational advantage of fast polynomial multiplication compared to slower matrix-vector multiplication in standard LWE.

5.1 The RLWE-FE scheme

Our proposed scheme is shown below. It allows to encrypt vectors from the l-dimensional space with absolute value of smaller than B_x while the coefficient of functional-vectors are bounded by B_y. We have postponed the parameter-setting to 5.2.

Construction:

- **Setup:** For n, a power of two, we fix a prime q and ring $\mathbb{R} = \mathbb{Z}[x]/(x^n + 1)$, i.e. the elements of $\mathbb{R}$ are polynomials of degree at most $n-1$ with coefficient in $\mathbb{Z}$. Denote with $\mathbb{R}_q = \mathbb{R}/(q\mathbb{Z})$ (meaning that the coefficient belong to $\mathbb{Z}_q$). Fix σ such that the ring-LWE decision problem is hard over $\mathbb{R}_q$, where the errors and secret are sampled from D_σ. We uniformly at random sample $a \in \mathbb{R}_q$ (sampling coefficient independently) and elements $\{s_i \mid i \in [l]\}, \{e_i \mid i \in [l]\}$ from $\mathbb{R}$, by sampling coefficients of s_i, e_i from the discrete Gaussian distribution with standard deviation σ_1, denoted by D_{σ_1}. Then $\{s_i \mid i \in [l]\}$ is a set of secret keys and the public key is $(a, \{pk_i \mid i \in [l]\})$, where $\mathsf{pk}_i = as_i + e_i \in \mathbb{R}_q$. Also we set $K > lB_xB_y$.
- **Encrypt:** To encrypt a vector $\boldsymbol{x} = (x_1, \ldots, x_l) \in \mathbb{Z}^l$ with $\|\boldsymbol{x}\| \leq B_x$ we sample polynomials r and f_0 by sampling their coefficients independently from D_{σ_2} , and $\{f_i \mid i \in [l]\}$ by sampling their coefficients independently from D_{σ_3}. We fix $1_{\mathbb{R}}$ to be the identity element of $\mathbb{R}_q$ (or it can be a polynomial of degree n with all coefficient equal $1 \in \mathbb{Z}_q$) and calculate:
$$\mathsf{ct}_0 = ar + f_0 \in \mathbb{R}_q,$$
$$\mathsf{ct}_i = \mathsf{pk}_i r + f_i + \lfloor q/K \rfloor x_i 1_R \in \mathbb{R}_q.$$
Then $(\mathsf{ct}_0, \{\mathsf{ct}_i\}_{i\in[l]})$ is the encryption of $\boldsymbol{x}$.
- **KeyGen:** To generate a key that decrypts $\langle \boldsymbol{x}, \mathbf{y} \rangle$ for $\mathbf{y} = (y_1, \ldots, y_n) \in \mathbb{Z}^l$ such that $\|y\| < B_y$, we calculate:
$$\mathsf{sk}_y = \sum_{i=1}^{l} y_i s_i \in \mathbb{R}.$$
- **Decryption:** To decrypt $(\mathsf{ct}_0, \{\mathsf{ct}_i\}_{i\in[l]})$ using sk_y and $\mathbf{y}$ we calculate:
$$d = (\sum_{i=1}^{l} y_i \mathsf{ct}_i) - \mathsf{ct}_0 \mathsf{sk}_y \mod \mathbb{R}_q.$$

Then d should be close to $\lfloor q/K \rfloor \langle \boldsymbol{x}, \mathbf{y} \rangle 1_R$ (a bit perturbed coefficients) and we can extract $\langle \boldsymbol{x}, \mathbf{y} \rangle$.

For security, we can show that the above scheme is *selectively secure against chosen-plaintext attacks* (sel-IND-CPA) under the assumption of hardness of decisional Ring-LWE problem as follows:

Let $n, q, \mathbb{R}_q, \sigma$ be such that the decisional Ring-LWE problem is hard if the coefficients of the secret and errors are sampled from a n-dimensional discrete Gaussian distributions D_σ. Then the inner product ring-LWE scheme with parameters $\sigma_1, \sigma_2, \sigma_3$ is sel-IND-FE-CPA secure for proper choice of parameters (see 5.2).

5.2 Parameter choice

The parameters should be chosen such that the scheme produces correct result. Below we show the constraints that must be satisfied while choosing the parameters.

Correctness. we have the following useful fact helping us to find the parameters for the correctness.

Lemma 1 ([35]). *For any $k > 0$, $\Pr_{x \leftarrow D_\sigma}[|x| > \sqrt{k}\sigma] \leq 2e^{-k/2}$.(one dimension Gaussian)*

Proof. We can write d as

$$\begin{aligned} d &= (\sum_{i=1}^{l} y_i \mathsf{ct}_i) - \mathsf{ct}_0 \mathsf{sk}_y \mod \mathbb{R}_q \\ &= \sum_i y_i(\mathsf{pk}_i r + f_i + \lfloor q/K \rfloor x_i 1_R) - (ar + f_0) \sum_i y_i s_i \\ &= \sum_i (y_i a s_i r + y_i e_i r + y_i f_i + \lfloor q/K \rfloor x_i y_i 1_R) - ar \sum_i y_i s_i - f_0 \sum_i y_i s_i \\ &= \sum_i (y_i e_i r + y_i f_i + f_0 y_i s_i) + \lfloor q/K \rfloor x_i y_i 1_R \\ &= \mathtt{noise} + \lfloor q/K \rfloor \langle x, y \rangle 1_R \end{aligned}$$

For the correctness we need $\|\mathtt{noise}\| < \lfloor q/2K \rfloor$. By lemma 1 for the security parameter k, with overwhelming probability we have, $\|e_i\|, \|s_i\| \leq \sqrt{k}\sigma_1$, also $\|r\|, \|f_0\| \leq \sqrt{k}\sigma_2$ and $\|f_i\| \leq \sqrt{k}\sigma_3$.Thus,

$$\|\sum_i \mathsf{k}_i(e_i r + f_i + f_0 s_i)\| < l(2nk\sigma_1\sigma_2 + \sqrt{k}\sigma_3)B_y$$

where $\|y\| < B_y$. Meaning that for the correctness we need

$$l(2nk\sigma_1\sigma_2 + \sqrt{k}\sigma_3)B_y < \lfloor q/2K \rfloor.$$

□

We used a python script and state-of-the-art security estimation techniques [2] to find some parameters form different values of B_x, B_y and l that satisfies all the constraints shown in Lemma 1. This is shown in Table. 5.

B_x, B_y, l	n, $\log q$	σ_1,	σ_2,	σ_3	PQ security
16, 16, 128	4096, 79	363.03,	168235570.5,	336471139	136
32, 32, 512	4096, 90	1449.1,	1343104100,	2686208198	113
64, 64, 1024	4096, 95	4097,	5370019841,	10740039680	105

Table 5: Suggested RLWE based FE parameters.

5.3 The CRT+NTT multiplication

As described in Sec. 5.2, the moduli q and the degree of polynomial is rather large. Considering most common 32 or 64 bit processors the elements of the underlying field $\mathbb{R}_q$ will not fit in a single word. In this scenario, the most naïve strategy is to use multi-precision arithmetic. However, it has been shown before [32] in the context of homomorphic encryption that this strategy leads to very poor efficiency. We use the same strategy adapted in homomorphic encryption. We break up each elements of $a \in \mathbb{Z}_q$ into two elements a_1, a_2 such that $a = a_1 \mod q_1$ and $a = a_2 \mod q_2$, where $q = q_1 q_2$. We perform all the operations in $\mathbb{R}_{q_1}$ and $\mathbb{R}_{q_2}$ instead of $\mathbb{R}_q$ and combine the results using Chinese remainder theorem (CRT) at the end using Garner's algorithm as shown in Alg. 6. This strategy is known as residual number system (RNS) and was first proposed by Bajard *et al.* [6] in the context of homomorphic encryption.

Algorithm 6: Garner's algorithm for Chinese remainder theorem

input : A positive integer $M = \prod_{i=1}^{t} m_i > 1$, with $\texttt{gcd}(m_i, m_j) = 1$ for all $i \neq j$ and $v(x) = (v_1, v_2, \cdots, v_t)$ such that $X \equiv v_i \mod m_i$ for all i.
output: x such that $X = x \mod M$

for $(i = 2; i \leq t; i++)$ **do**
  $C_i = 1$;
  for $j = 1; j \leq i-1; j++$ **do**
    $u = m_j^{-1} \mod m_i$;
    $C_i = u \cdot C_i \mod m_i$;
  $u = v_1$; $x = u$;
  for $(i = 2; i \leq t; i++)$ **do**
    $u = (v_i - x) \cdot C_i \mod m_i$;
    $x = x + u \cdot \prod_{j=1}^{i-1} m_j$
return x

For individual multiplications in $\mathbb{R}_{q_1}$ or $\mathbb{R}_{q_2}$ we choose number theoretic transform (NTT) based polynomial multiplication which has $O(n \log n)$ complexity. We have seen in Sec. 4, than Toom-Cook based multiplication is very efficient when multiplying two polynomial of small degree e.g. 255 although having a $O(n^2)$ complexity. However, as we are dealing with polynomials with very high degree e.g. 4095 in our RLWE based FE, the advantages of regular memory access and faster modular reduction are eclipsed by a slower quadratic complexity.

NTT polynomial multiplication is a specialized version of the discrete Fourier transform. To apply NTT based polynomial multiplication we need our n to be a power of 2 and the modulus q to be a prime such that $q \equiv 1 \mod 2n$. Let $a, b \in \mathbb{R}_q$ be two polynomials and $c = a \times b \in \mathbb{R}_q$ be their ring product. Also, let $a = (a[0], a[1], \cdots, a[n-1])$ and $b = (b[0], b[1], \cdots, b[n-1])$ be

the coefficient array of the polynomials a and b respectively. We denote ψ as the primitive $2n$-th root of unity in $\mathbb{Z}_q$ *i.e.* $\psi^{2n} \equiv 1 \mod q$. Such root is guaranteed to exist due to our choice of the prime. Two main components of NTT multiplication are forward NTT (NTT) and reverse NTT (INTT). The forward NTT transformation $\tilde{a} = \texttt{NTT}(a)$ is defined as $\tilde{a}[i] = \sum_{j=0}^{n-1} a[j]\psi^{ij} \mod q$ for $i = 0, 1, \cdots, n-1$. The inverse transformation is given by $a = INTT(\tilde{a}) = n^{-1} \sum_{j=0}^{n-1} \tilde{a}[j]\psi^{ij} \mod q$. It should be noted that $a = INTT(NTT(a))$. The polynomial multiplication using NTT is achieved by $c = a \times b = INTT(NTT(a) \circ NTT(b))$ where $\circ$ denotes the component-wise multiplication of two vectors.

Algorithm 7: Forward NTT transformation using Cooley-Tukey method

input : A vector $a = (a[0], a[1], \cdots, a[n-1]) \in \mathbb{Z}_n^{q'}$ in standard ordering, where q' is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\psi_{rev} \in \mathbb{Z}_{q'}^n$ storing powers of ψ in a bit-reversed order

output: $a \leftarrow \texttt{NTT}(a)$

```
t = n;
for (m = 1; m < n; m = 2m) do
    t = t/2;
    for (i = 0; i < m; i++) do
        j1 = 2 · i · t;
        j2 = j1 + t − 1;
        S = ψ_rev[m + i];
        for (j = j1; j ≤ j2; j++) do
            U = a[j];
            V = a[j + t] · S;
            a[j] = U + V  mod q';
            a[j + t] = U − V  mod q';
return a
```

Almost all forward or inverse NTT transformations [40, 46, 33] require the input array to be arranged in standard array and produce results in a bit-reversed ordering[2] or accepts the array in bit-reversed ordering and produces result in standard ordering. This requires a bit-reversal permutation on the input or output array before or after each transformations. For our parameters performing bit-reversal steps after each transformation is costly. To solve this problem we took an approach shown in [34, 42]. We use the Cooley-Tukey (Alg. 7) transformation for our forward NTT operation. This method accepts the input in standard ordering but produces the results in bit-reversed ordering. We perform the component-wise multiplication $\circ$ in this ordering. Finally, for the inverse NTT operation we use the Gentleman-Sande (Alg. 8) transformation which accepts inputs in the bit-reversed ordering and produces output in the standard ordering. Therefore, using Cooley-Tukey and Gentleman-Sande transformations in conjunction for forward and reverse NTT transform we can eliminate the need for costly bit-reversal permutations.

[2] https://en.wikipedia.org/wiki/Bit-reversal_permutation

Algorithm 8: Inverse NTT transformation using Gentleman-Sade method

input : A vector $a = (a[0], a[1], \cdots, a[n-1]) \in \mathbb{Z}_n^{q'}$ in bit-reversed ordering, where q' is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two. A precomputed table $\psi_{rev}^{-1} \in \mathbb{Z}_{q'}^n$ storing powers of ψ^{-1} in bit-reversed order

output: $a \leftarrow \texttt{INTT}(a)$

$t = 1$;
for $(m = n; m > 1; m = m/2)$ **do**
  $j_1 = 0$;
  $h = m/2$;
  for $(i = 0; i < h; i++)$ **do**
    $j_2 = j_1 + t - 1$;
    $S = \psi_{rev}^{-1}[h + i]$;
    for $(j = j_1; j \leq j_2; j++)$ **do**
      $U = a[j]$;
      $V = a[j + t] \cdot S$;
      $a[j] = U + V \mod q'$;
      $a[j + t] = (U - V) \cdot S \mod q'$;
    $j_1 = j_1 + 2t$;
  $t = 2t$;
for $(j = 0; j < n; j++)$ **do**
  $a[j] = a[j] \cdot n^{-1} \mod q$;
return a

6 Conclusions

In this deliverable we have discussed implementations of building blocks for lattice-based FE schemes that incorporate built-in resistance against timing attacks. In particular, we have presented a method to sample from Gaussian distributions and a technique to perform polynomial multiplication. Both implementations have been designed to be constant-time, and optimized (where possible) to achieve competitive performances. Here in this document we also presented a fast HW-SW codesign approach for polynomial multiplication. We have illustrated this by comparing our results with those from the state-of-the-art.

Further, we have presented parameters and overview of our ring learning with errors based functional encryption scheme. We presented methods for optimal implementation of our functional encryption scheme.

The proposed implementations represent the first step towards developing FE implementations in a HW-operated setting. Variations in execution time are one of the simplest and most exploited type of side-channel leakages in the literature. Time-constant building blocks are therefore the natural starting point before considering more powerful adversaries, e.g. capable of exploiting other side-channel leakage sources such as power consumption or electromagnetic emanations. Finally, we presented a FE scheme which is, to the best of our knowledge, the first one to be based on ring-learning with errors. Such schemes are more efficient compared to their standard lattice based counterparts due to the use of faster polynomial arithmetic. We also propose carefully chosen parameters and polynomial multiplication strategy to facilitate faster NTT based polynomial multiplication in our schemes. We believe FE can be made very practical to solve real-world problems using the strategies proposed in this document.

Bibliography

[1] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. Cryptology ePrint Archive, Report 2015/017, 2015. https://eprint.iacr.org/2015/017. (Page 33)

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046. (Page 34)

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy - S&P 2013*, pages 526–540. IEEE Computer Society, 2013. (Page 9)

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security Symposium - USENIX 2016*, pages 327–343. USENIX Association, 2016. (Page 10)

[5] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. NewHope on ARM Cortex-M. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - SPACE 2016*, volume 10076 of *Lecture Notes in Computer Science*, pages 332–349. Springer, 2016. (Pages 25 and 26)

[6] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. Cryptology ePrint Archive, Report 2016/510, 2016. https://eprint.iacr.org/2016/510. (Page 35)

[7] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):17–61, 2019. (Pages 28 and 32)

[8] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006. (Page 8)

[9] Daniel J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, April 2005. (Page 9)

[10] Eli Biham. A Fast New DES Implementation in Software. In Eli Biham, editor, *Fast Software Encryption - FSE '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997. (Page 14)

[11] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *Security and Privacy - S&P 2015*, pages 553–570. IEEE Computer Society, 2015. (Pages 10 and 13)

[12] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *European Symposium on Security and Privacy - EuroS&P 2018*, pages 353–367. IEEE, 2018. (Pages 10 and 26)

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. (Page 9)

[14] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. Lizard: Cut Off the Tail! A Practical Post-quantum Public-Key Encryption from LWE and LWR. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks - SCN 2018*, volume 11035 of *Lecture Notes in Computer Science*, pages 160–177. Springer, 2018. (Page 10)

[15] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 206–214, 2019. (Pages 31 and 32)

[16] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018. (Pages 10, 18, 19, 20, and 21)

[17] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *Design, Automation & Test in Europe - DATE 2015*, pages 339–344. ACM, 2015. (Pages 25 and 26)

[18] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013. (Page 10)

[19] Léo Ducas and Phong Q. Nguyen. Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2012. (Page 10)

[20] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng. Commun. Comput.*, 25(3):159–180, 2014. (Page 11)

[21] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001. (Page 8)

[22] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):372–393, 2018. (Page 32)

[23] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of USSR Academy of Sciences*, 145(7):293–294, 1962. (Pages 19 and 22)

[24] Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM cca-secure module lattice-based key encapsulation on ARM. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):243–266, 2018. (Pages 19, 21, and 30)

[25] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-Time Discrete Gaussian Sampling. *IEEE Trans. Computers*, 67(11):1561–1571, 2018. (Pages 10, 12, 15, 16, and 17)

[26] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete gaussian sampling. A case study on the falcon signature scheme. In *Design Automation Conference - DAC 2019*, pages 88:1–88:6. ACM, 2019. (Pages 10 and 13)

[27] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976. (Pages 10 and 11)

[28] Donald Knuth. *The Art of Computer Programming, Volume 2. Third Edition.* Addison-Wesley, 1997. (Page 20)

[29] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. (Pages 8 and 9)

[30] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. (Page 8)

[31] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on fpgas. Cryptology ePrint Archive, Report 2017/690, 2017. https://eprint.iacr.org/2017/690. (Page 32)

[32] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. Cryptology ePrint Archive, Report 2014/062, 2014. https://eprint.iacr.org/2014/062. (Page 35)

[33] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. Cryptology ePrint Archive, Report 2015/410, 2015. https://eprint.iacr.org/2015/410. (Page 36)

[34] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Report 2016/504, 2016. https://eprint.iacr.org/2016/504. (Page 36)

[35] Vadim Lyubashevsky. Lattice signatures without trapdoors. pages 738–755, 2012. (Page 34)

[36] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism. *IACR Cryptol. ePrint Arch.*, 2020:321, 2020. (Pages 19 and 32)

[37] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices. In *Design Automation Conference 2014 - DAC '14*, pages 110:1–110:6. ACM, 2014. (Pages 25 and 26)

[38] Chris Peikert. An Efficient and Parallel Gaussian Sampler for Lattices. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010. (Page 10)

[39] Colin Percival. Cache missing for fun and profit. Proceedings of BSDCan, 2005. (Page 9)

[40] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, pages 68–85, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (Page 36)

[41] pqm4. Post-quantum crypto library for the arm cortex-m4. https://github.com/mupq/pqm4, 2018. [Online; accessed 15-July-2019]. (Pages 25 and 26)

[42] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. Cryptology ePrint Archive, Report 2015/382, 2015. https://eprint.iacr.org/2015/382. (Page 36)

[43] Jean-Jacques Quisquater and David Samyde. Eddy current for Magnetic Analysis with Active Sensor. In *Esmart 2002*, 2002. (Page 8)

[44] Daniel S. Roche. Space- and time-efficient polynomial multiplication. In *Symbolic and Algebraic Computation, International Symposium - ISSAC 2009*, pages 295–302, 2009. (Pages 19 and 20)

[45] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel resistant discrete gaussian sampling. Cryptology ePrint Archive, Report 2014/591, 2014. https://eprint.iacr.org/2014/591.pdf. (Page 10)

[46] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe based cryptoprocessor. Cryptology ePrint Archive, Report 2013/866, 2013. https://eprint.iacr.org/2013/866. (Page 36)

[47] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. High Precision Discrete Gaussian Sampling on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013*, volume 8282 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2013. (Page 12)

[48] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002. (Page 8)