Sapphire-Sim: Macro-Op-Level Simulator for Ring-LWE and Module-LWE Hardware Acceleration

Utsav Banerjee and Anantha P. Chandrakasan

Massachusetts Institute of Technology, Cambridge, MA, USA
utsav@mit.edu

Abstract. This manuscript describes a Python-based open-source cycle-accurate simulator for the Sapphire lattice-crypto processor which can be used to profile the performance of Ring-LWE and Module-LWE algorithms. This allows fast evaluation of lattice-based protocols with varying parameter choices but without any hardware design effort, which is especially important for a fast evolving field such as lattice-based cryptography. The simulator not only reports accurate cycle counts and execution times but also macro-operation-level power and average energy consumption modelled using measurements from the Sapphire test chip at various operating conditions. Detailed description of the custom instructions, simulation options and example code are also provided as reference.

Keywords: Lattice-based Cryptography · Ring-LWE · Module-LWE · post-quantum · Number Theoretic Transform · Sampling · hardware implementation · simulator.

1 Introduction

Lattice-based cryptography has emerged as a prime candidate for computationally efficient post-quantum public key encryption, key encapsulation and digital signatures, and also allows the construction of novel cryptographic primitives such as homomorphic encryption and functional encryption [1]. In the light of recent advancements in quantum computing technology, NIST is now standardizing post-quantum cryptographic (PQC) algorithms [2], and lattice-based cryptography accounts for 53% (9 out of 17) of the public key encryption and key encapsulation schemes and 33% (3 out of 9) of the signature schemes among the NIST PQC Round 2 candidates [3].

The theoretical foundation of several of these lattice-based protocols lies in the *learning with errors* (LWE) problem and its variants such as Ring-LWE and Module-LWE, and their hardness has been well-studied in the presence of both classical and quantum adversaries [1]. Since lattice-based cryptography is a fast evolving field, the associated parameters, such as dimension n, modulus q, choice of error distribution and standard deviation σ, etc, vary widely among different algorithms and protocols. To allow the flexibility to implement different such parameter choices, a configurable lattice-crypto processor – "Sapphire" – was presented in [4]. A Python-based cycle-accurate simulator for the Sapphire crypto-processor is described here, which can be used to profile the power consumption and performance of algorithms based on Ring-LWE and Module-LWE.

Section 2 provides a brief overview of the hardware architecture and supported parameters. Section 3 describes all the instructions and their usage in detail, along with example code snippets in Section 4. Section 5 describes how to run the simulator with various simulation options. Finally, Section 6 provides simulation results for the CPA-secure public key encryption (CPA-PKE) schemes NewHope [5], Kyber-v1 [6], R.EMBLEM [7] and LIMA [8]. The simulator is available on GitHub[1].

[1]https://github.com/banerjeeutsav/sapphire_sim

2 Hardware Architecture and Supported Parameters

The architecture of the Sapphire lattice-crypto core is shown in Fig. 1. A pair of 12 KB *Polynomial Caches* interface with a configurable *Modular Arithmetic Unit* to perform number-theoretic transform (NTT) and polynomial arithmetic. Each 12 KB polynomial cache consists of four 1024×24-bit single-port SRAMs. Together, the 24 KB cache stores 8192 24-bit entries, which can be split into four 2048-dimension polynomials or eight 1024-dimension polynomials or sixteen 512-dimension polynomials or thirty-two 256-dimension polynomials or sixty-four 128-dimension polynomials or one-hundred-twenty-eight 64-dimension polynomials. The modular arithmetic unit consists of a butterfly module, with modular multiplier, adder and subtractor with 24-bit data-path and configurable modulus. The butterfly module supports both the Cooley-Tukey (decimation-in-time or DIT) and the Gentleman-Sande (decimation-in-frequency or DIF) configurations. A 15 KB *NTT Constants RAM* stores the pre-computed twiddle factors. Apart from butterfly and modular arithmetic, additional circuitry is also present to perform bit-wise AND, OR, XOR, left shift and right shift operations. An energy-efficient *Keccak-f[1600] Core*, used for hashing and pseudo-random number generation (PRNG), drives the *Discrete Distribution Sampler*. The processor is equipped with a 1 KB instruction memory and custom 32-bit instructions are used to implement various lattice-based algorithms. In the test chip, the Sapphire crypto core is integrated with a RISC-V micro-processor through its memory-mapped interface. However, the simulator currently replicates functionality of the crypto core only, that is, neither RISC-V programs nor data movement between the RISC-V processor and the crypto core can be simulated. Please note that this is not an architectural simulator, that is, it is only functionally correct and does not replicate any internal circuitry of the crypto core (unlike an HDL-based RTL simulation).

The simulator currently supports polynomial dimension $n \in \{64, 128, 256, 512, 1024, 2048\}$ and prime modulus $q \in \{3329, 7681, 12289, 40961, 65537, 120833, 133121, 184321, 4205569, 4206593, 8058881, 8380417, 8404993\}$. Polynomials can be sampled from the following discrete probability distributions: (1) binomial sampling with standard deviation $\sigma = \sqrt{k/2}$ for $k \in (0, 32]$, (2) inversion sampling from the cumulative distribution table of a discrete symmetric zero-mean distribution with support $s < 64$ and precision $r \leq 32$ bits, (3) rejection sampling in $\mathbb{Z}_q$, (4) uniform sampling in $[-\eta, \eta]$ for $\eta < q$, (5) trinary sampling in $\{-1, 0, +1\}$ with m non-zero coefficients $(m < n)$, (6)

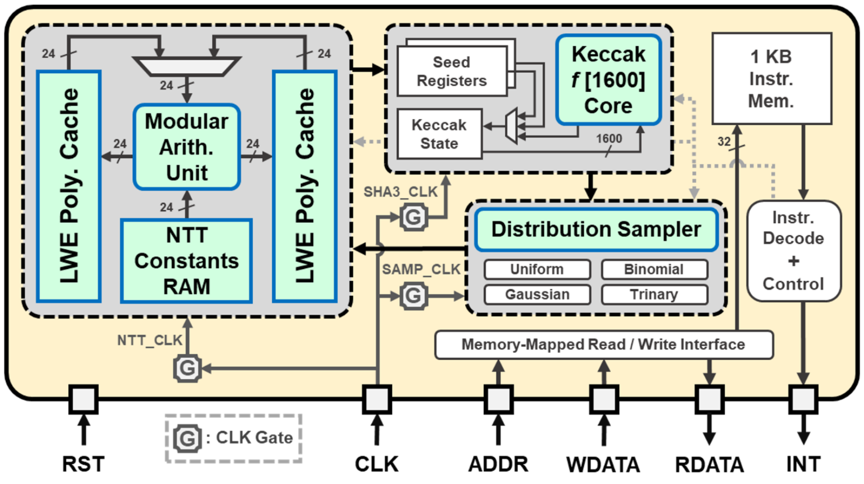


Fig. 1. Architecture of the Sapphire lattice-crypto core.

trinary sampling in $\{-1, 0, +1\}$ with m_0 +1's and m_1 −1's ($m_0 < n$, $m_1 < n$ and $m_0+m_1 < n$), and (7) trinary sampling in $\{-1, 0, +1\}$ with coefficients distributed as $\Pr[x = 1] = \Pr[x = -1] = \rho/2$ and $\Pr[x = 0] = 1 - \rho$ for $\rho \in \{1/2, 1/4, 1/8, \cdots, 1/128\}$. All hardware implementation details are available in [4]. Although the 1 KB instruction memory puts a limit on the program size, the simulator currently allows arbitrarily large programs.

3 Summary of Instructions

All custom instructions supported by the crypto-processor (and the simulator) are summarized here. The polynomials are accessed as `poly = #pnum` (where `#pnum` $< 8192/n$). For operations that involve a pair of polynomials, the source polynomial is specified as `poly_src` and the destination polynomial is specified as `poly_dst`. For such operations, except `poly_copy`, it must be ensured that (`poly_src` $< 4096/n$, `poly_dst` $\geq 4096/n$) or (`poly_dst` $< 4096/n$, `poly_src` $\geq 4096/n$). Apart from the polynomials, the following internal registers can also be manipulated:

- 256-bit seed registers `r0` and `r1`
- 24-bit temporary registers `reg` / `tmp`
- 16-bit counter registers `c0` / `c1`
- 2-bit `flag` register used to store comparison results (-1, 0 or +1)

The `tmp` register is clobbered by many of the instructions, so it should be used carefully. Following is the list of instructions along with their description and usage:

Parameter Configuration:

```
config (n, q)
```

used to configure parameters n and q; must be first instruction of the program; supported parameters are $n \in \{64, 128, 256, 512, 1024, 2048\}$ and $q \in \{3329, 7681, 12289, 40961, 65537, 120833, 133121, 184321, 4205569, 4206593, 8058881, 8380417, 8404993\}$

Register Assignment / Increment / Decrement:

```
c0 = #VAL / c0 + #VAL / c0 - #VAL
c1 = #VAL / c1 + #VAL / c1 - #VAL
reg = #VAL / tmp
tmp = #VAL
```

where `#VAL` is unsigned integer of appropriate size, that is, `#VAL` $\in [0, 2^{16})$ for `c0` / `c1` and `#VAL` $\in [0, 2^{24})$ for `reg` / `tmp`; the register increment and decrement computations are performed modulo 2^{16}.

Register Arithmetic and Logic Operations:

```
tmp = tmp (OP) reg
```

where `(OP)` can be + (addition modulo q), − (subtraction modulo q), ∗ (multiplication modulo q), & (bitwise AND), | (bitwise OR), ∧ (bitwise XOR), >> (right shift) or << (left shift); the shift operations output `0x000000` for `reg` ≥ 24.

Polynomial Initialization:

```
init (poly)
```

initializes the polynomial with all zero coefficients.

Polynomial Maximum of Coefficients:

`reg = max (poly)`

the maximum is computed over polynomial coefficients a_i, with $i \in [0, \cdots, n)$, as $\max\limits_i |a_i \bmod^{\pm} q|$, where $x' = x \bmod^{\pm} q$ is defined as the unique element x' in the range $[-\frac{q-1}{2}, \frac{q-1}{2}]$ such that $x' = x \bmod q$; the final result is in $[0, q)$; note that register `reg` is clobbered at the end of this operation.

Polynomial Sum of Coefficients:

`reg = sum (poly)`

the sum is computed over polynomial coefficients a_i, with $i \in [0, \cdots, n)$, as $(\sum\limits_{i=0}^{n-1} a_i \bmod^{\pm} q) \bmod q$, where $x' = x \bmod^{\pm} q$ is defined as the unique element x' in the range $[-\frac{q-1}{2}, \frac{q-1}{2}]$ such that $x' = x \bmod q$; the final result is in $[0, q)$; note that register `reg` is clobbered at the end of this operation.

Register-Polynomial Operations:

`reg = (poly)[#VAL] / (poly)[c0] / (poly)[c1]`

`(poly)[#VAL] / (poly)[c0] / (poly)[c1] = reg`

where the polynomial coefficient index must be in the appropriate range, that is, `#VAL` $< n$, and `c0` / `c1` are internally reduced modulo n to derive the index.

Polynomial Number Theoretic Transform:

`transform (mode, poly_dst, poly_src)`

where `mode` $\in$ {`DIF_NTT, DIF_INTT, DIT_NTT, DIT_INTT`}; ensure that (`poly_src` $< 4096/n$, `poly_dst` $\geq 4096/n$) or (`poly_dst` $< 4096/n$, `poly_src` $\geq 4096/n$); must have $q \equiv 1 \bmod 2n$; note that source polynomial `poly_src` is clobbered at the end of this operation.

Polynomial Pre/Post-Processing for Negative Wrapped Convolution:

`mult_psi (poly) / mult_psi_inv (poly)`

pre- and post- processing involve multiplying the i-th coefficient by $\psi^i \bmod q$ and $n^{-1}\psi^{-i} \bmod q$ respectively, where ψ is the $2n$-th root of unity modulo q (ψ exists only if $q \equiv 1 \bmod 2n$); the simulator maintains a lookup table of ω and ψ, respectively the n-th and $2n$-th roots of unity modulo q.

Binomial Sampling:

`bin_sample (prng, seed, k, poly)`

`bin_sample (prng, seed, c0, c1, k, poly)`

generates polynomial coefficients from binomial distribution with standard deviation $\sigma = \sqrt{\texttt{k}/2}$ for `k` $\in (0, 32]$; `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}.

Cumulative Distribution Table Inversion Sampling:

`cdt_sample (prng, seed, r, poly)`

`cdt_sample (prng, seed, c0, c1, r, poly)`

generates polynomial coefficients from discrete symmetric zero-mean distribution with precision `r` ≤ 32 bits; the cumulative distribution table (CDT) is provided separately (discussed in Section 5) and the distribution support s is inferred from the same; `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}.

Rejection Sampling:
`rej_sample (prng, seed, poly)`
`rej_sample (prng, seed, c0, c1, poly)`
generates polynomial coefficients distributed uniformly in $\mathbb{Z}_q$ by rejection sampling; `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}.

Uniform Sampling:
`uni_sample (prng, seed, eta, poly)`
`uni_sample (prng, seed, c0, c1, eta, poly)`
generates polynomial coefficients distributed uniformly in [`-eta`, `+eta`] by rejection sampling (`eta` $< q$); `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}; note that register `reg` is clobbered at the end of this operation.

Trinary Sampling (1):
`tri_sample_1 (prng, seed, m, poly)`
`tri_sample_1 (prng, seed, c0, c1, m, poly)`
generates trinary polynomial with `m` non-zero coefficients (`m` $< n$); `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}.

Trinary Sampling (2):
`tri_sample_2 (prng, seed, m0, m1, poly)`
`tri_sample_2 (prng, seed, c0, c1, m0, m1, poly)`
generates trinary polynomial with `m0` $+1$'s and `m1` -1's (`m0` $< n$, `m1` $< n$ and `m0` $+$ `m1` $< n$); `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}; note that register `reg` is clobbered at the end of this operation.

Trinary Sampling (3):
`tri_sample_3 (prng, seed, rho, poly)`
`tri_sample_3 (prng, seed, c0, c1, rho, poly)`
generates trinary polynomial with coefficients distributed as $\Pr[x = 1] = \Pr[x = -1] =$ `rho`$/2$ and $\Pr[x = 0] = 1 -$ `rho` for `rho` $\in \{$`1/2`, `1/4`, `1/8`, $\cdots$, `1/128`$\}$; `c0` / `c1` are used as counters for sampling multiple polynomials from same seed; `c0` / `c1` can be either specified separately using register assignment instructions or provided as arguments of the sampling instruction (both cases function exactly the same internally); `prng` $\in$ {`SHAKE-128, SHAKE-256`} and `seed` $\in$ {`r0, r1`}.

Polynomial Copy:
`poly_copy (poly_dst, poly_src)`
copies source polynomial `poly_src` into destination polynomial `poly_dst`; copying is significantly faster if (`poly_src` $< 4096/n$, `poly_dst` $\geq 4096/n$) or (`poly_dst` $< 4096/n$, `poly_src` $\geq 4096/n$).

Polynomial Arithmetic and Logic Operations:

`poly_op (op, poly_dst, poly_src)`

where `op` can be one of the following: {`ADD, SUB, MUL, BITREV, CONST_ADD, CONST_SUB, CONST_MUL, CONST_AND, CONST_OR, CONST_XOR, CONST_RSHIFT, CONST_LSHIFT`}, described below in terms of `i`-th coefficients of the source and destination polynomials:

`ADD`:	`poly_dst[i]` = (`poly_src[i]` + `poly_dst[i]`) mod q
`SUB`:	`poly_dst[i]` = (`poly_src[i]` − `poly_dst[i]`) mod q
`MUL`:	`poly_dst[i]` = (`poly_src[i]` × `poly_dst[i]`) mod q
`BITREV`:	`poly_dst[i]` = `poly_src[BitRev(i)]`
`CONST_ADD`:	`poly_dst[i]` = (`poly_src[i]` + `reg`) mod q
`CONST_SUB`:	`poly_dst[i]` = (`poly_src[i]` − `reg`) mod q
`CONST_MUL`:	`poly_dst[i]` = (`poly_src[i]` × `reg`) mod q
`CONST_AND`:	`poly_dst[i]` = `poly_src[i]` & `reg`
`CONST_OR`:	`poly_dst[i]` = `poly_src[i]` $\vert$ `reg`
`CONST_XOR`:	`poly_dst[i]` = `poly_src[i]` $\wedge$ `reg`
`CONST_RSHIFT`:	`poly_dst[i]` = `poly_src[i]` >> `reg`
`CONST_LSHIFT`:	`poly_dst[i]` = `poly_src[i]` << `reg`

the shift operations on each coefficient output `0x000000` for `reg` ≥ 24; ensure that (`poly_src` $< 4096/n$, `poly_dst` $\geq 4096/n$) or (`poly_dst` $< 4096/n$, `poly_src` $\geq 4096/n$).

Polynomial Shift:

`shift_poly (ring, poly_dst, poly_src)`

circular left shift of the polynomial coefficients, that is, multiplication by x in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ or $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n-1)$; `ring` can be either `x^N+1` or `x^N-1`; ensure that (`poly_src` $< 4096/n$, `poly_dst` $\geq 4096/n$) or (`poly_dst` $< 4096/n$, `poly_src` $\geq 4096/n$).

Polynomial Equality Check:

`flag = eq_check (poly0, poly1)`

checks whether the two polynomials `poly0` and `poly1` are equal; `flag` is 1 or 0 depending on whether the equality check has passed or failed respectively; ensure that (`poly0` $< 4096/n$, `poly1` $\geq 4096/n$) or (`poly1` $< 4096/n$, `poly0` $\geq 4096/n$); note that register `flag` is clobbered at the end of this operation.

Polynomial Infinity Norm Check:

`flag = inf_norm_check (poly, bound)`

checks whether infinity norm of the polynomial is not larger than `bound` (`bound` $< 2^{24}$), that is, the check fails when $a_i \in (\texttt{bound}, q - \texttt{bound})$ for at least one polynomial coefficient a_i, with $i \in [0, \cdots, n)$; `flag` is 1 or 0 depending on whether the infinity norm check has passed or failed respectively; note that registers `reg` and `flag` are clobbered at the end of this operation.

Register-Value Comparison:

`flag = compare (reg / tmp / c0 / c1, #VAL)`

compares one of the registers `reg` / `tmp` / `c0` / `c1` with unsigned integer `#VAL` of appropriate size, that is, `#VAL` $\in [0, 2^{16})$ for `c0` / `c1` and `#VAL` $\in [0, 2^{24})$ for `reg` / `tmp`; `flag` is -1, 0 and +1 for the register comparison result being "lesser than", "equal to" and "greater than" respectively; note that register `flag` is clobbered at the end of this operation.

Branching:
`if (flag == / != -1 / 0 / +1) goto <label>`
branches to `<label>` (labels discussed in Section 4) depending on the value of `flag`; typically follows polynomial equality check, polynomial infinity norm check or register-value comparison.

SHA-3 State Initialization:
`sha3_init`
initializes the SHA-3 internal state.

SHA-3 Register Absorption:
`sha3_256_absorb (r0 / r1)`
`sha3_512_absorb (r0 / r1)`
absorbs one of the 256-bit seed registers `r0` / `r1` for SHA3-256 or SHA3-512.

SHA-3 Polynomial Absorption:
`sha3_256_absorb (poly)`
`sha3_512_absorb (poly)`
absorbs each polynomial coefficient, padded to 32 bits, for SHA3-256 or SHA3-512.

SHA-3 Digest:
`r0 / r1 = sha3_256_digest`
`r0 || r1 = sha3_512_digest`
generates 256-bit SHA3-256 digest, stored in either `r0` or `r1`, or 512-bit SHA3-512 digest, stored together in `r0` and `r1`; note that registers `r0` or `r1` or both are clobbered at the end of this operation.

No Operation:
`nop`
does nothing.

End of Program:
`end`
marks the end of program; must be last instruction of the program.

The following debug instructions are also supported by the simulator to display intermediate results, load program inputs and save program outputs, and handle some commonly used encodings for mapping byte-strings to polynomials and vice-versa.

Print Register or Polynomial:
`print (r0 / r1)`
`print (reg / tmp / c0 / c1)`
`print (flag)`
`print (poly)`
prints the contents of 256-bit seed registers `r0 / r1` (in hexadecimal), 24-bit temporary registers `reg / tmp` (as unsigned integer), 16-bit counter registers `c0 / c1` (as unsigned integer), 2-bit flag register `flag` (as signed integer) or polynomials (list of n coefficients as unsigned integers reduced modulo q); works only when the `--verbose` option is enabled in the simulator (see Section 5).

Load or Save Seed Register:

`load (r0 / r1 , "<numpy_file_path>")`

`save (r0 / r1 , "<numpy_file_path>")`

loads / saves 256-bit seed register values from / to specified NumPy array file (must have ".npy" extension); by default, adds 2 cycles per 32-bit read / write and corresponding power consumption (to simulate the crypto-processor interface); can skip cycle and power overheads using the `--free_rw` simulator option (see Section 5).

Load or Save Polynomial:

`load (poly , "<numpy_file_path>")`

`save (poly , "<numpy_file_path>")`

loads / saves polynomial coefficient values from / to specified NumPy array file (must have ".npy" extension); by default, adds 2 cycles per coefficient read / write and corresponding power consumption (to simulate the crypto-processor interface); can skip cycle and power overheads using the `--free_rw` simulator option (see Section 5).

Generate Random Value for Seed Register:

`random (r0 / r1)`

generates random 256-bit value and writes to seed register; by default, adds 2 cycles per 32-bit write and corresponding power consumption (to simulate the crypto-processor interface); can skip cycle and power overheads using the `--free_rw` simulator option (see Section 5).

Generate Random Coefficients for Polynomial:

`random (poly, encoding , "<numpy_file_path>")`

generates random polynomial, e.g., message to be encrypted, with specified `encoding` $\in$ {`BINARY_0RED, BINARY_2RED, BINARY_4RED, BINARY_8RED, TRUNC_256, TRUNC_256_MSB`}:

`BINARY_0RED`:	generates n random bits r_i and computes polynomial coefficients a_i as: $a_i = \lfloor q/2 \rceil\, r_i$ for $i \in [0, n)$
`BINARY_2RED`:	generates $n/2$ random bits r_i and computes polynomial coefficients a_i as: $a_i = a_{i+n/2} = \lfloor q/2 \rceil\, r_i$ for $i \in [0, n/2)$
`BINARY_4RED`:	generates $n/4$ random bits r_i and computes polynomial coefficients a_i as: $a_i = a_{i+n/4} = a_{i+n/2} = a_{i+3n/4} = \lfloor q/2 \rceil\, r_i$ for $i \in [0, n/4)$
`BINARY_8RED`:	generates $n/8$ random bits r_i and computes polynomial coefficients a_i as: $a_i = a_{i+n/8} = \cdots = a_{i+3n/4} = a_{i+7n/8} = \lfloor q/2 \rceil\, r_i$ for $i \in [0, n/8)$
`TRUNC_256`:	generates 256 random bits r_i and computes polynomial coefficients a_i as: $a_i = \lfloor q/2 \rceil\, r_i$ for $i \in [0, 256)$ and $a_i = 0$ for $i \in [256, n)$
`TRUNC_256_MSB`:	generates 256 random bits r_i and computes polynomial coefficients a_i as: $a_i = (2r_i + 1) \cdot 2^{\lfloor \log_2 q \rfloor - 2}$ for $i \in [0, 256)$ and $a_i = 0$ for $i \in [256, n)$

saves polynomial coefficient values to specified NumPy array file (must have ".npy" extension); by default, adds 2 cycles per coefficient write and corresponding power consumption (to simulate the crypto-processor interface); can skip cycle and power overheads using the `--free_rw` simulator option (see Section 5).

Encode and Print Polynomial:

`encode_print (poly, encoding)`

encodes polynomial coefficients into bits and prints the corresponding byte-array with specified `encoding` $\in$ {`BINARY_0RED`, `BINARY_2RED`, `BINARY_4RED`, `BINARY_8RED`, `TRUNC_256`, `TRUNC_256_MSB`, `RECON_SIMPLE`}:

`BINARY_0RED`:	converts polynomial coefficients a_i into bits r_i as: $r_i = \lfloor (2/q)\, a_i \rceil$ for $i \in [0, n)$
`BINARY_2RED`:	converts polynomial coefficients a_i into bits r_i as: $t_i = \sum_{j=0}^{1} (a_{i+jn/2} - \lfloor q/2 \rfloor)$ $r_i = 0$ if $t_i > \lfloor q/2 \rceil$ and $r_i = 1$ otherwise for $i \in [0, n/2)$
`BINARY_4RED`:	converts polynomial coefficients a_i into bits r_i as: $t_i = \sum_{j=0}^{3} (a_{i+jn/4} - \lfloor q/2 \rfloor)$ $r_i = 0$ if $t_i > q$ and $r_i = 1$ otherwise for $i \in [0, n/4)$
`BINARY_8RED`:	converts polynomial coefficients a_i into bits r_i as: $t_i = \sum_{j=0}^{7} (a_{i+jn/8} - \lfloor q/2 \rfloor)$ $r_i = 0$ if $t_i > 2q$ and $r_i = 1$ otherwise for $i \in [0, n/8)$
`TRUNC_256`:	converts polynomial coefficients a_i into bits r_i as: $r_i = \lfloor (2/q)\, a_i \rceil$ for $i \in [0, 256)$
`TRUNC_256_MSB`:	converts polynomial coefficients a_i into bits r_i as: $r_i = \lfloor a_i / 2^{\lfloor \log_2 q \rfloor - 1} \rfloor$ for $i \in [0, 256)$
`RECON_SIMPLE`:	converts polynomial coefficients a_i into bits r_i as: $r_i = 0$ if $a_i \in [0, \lfloor q/4 \rceil) \cup (\lfloor 3q/4 \rceil, q)$ and $r_i = 1$ otherwise for $i \in [0, n)$

works only when the `--verbose` option is enabled in the simulator (see Section 5).

Encode and Compare Polynomials:

`encode_compare ("<numpy_file_path>", "<numpy_file_path>", encoding)`

loads polynomial coefficient values from specified pair of NumPy array files (both must have ".npy" extension), encodes them into bits and compares the corresponding byte-arrays with specified `encoding` $\in$ {`BINARY_0RED`, `BINARY_2RED`, `BINARY_4RED`, `BINARY_8RED`, `TRUNC_256`, `TRUNC_256_MSB`, `RECON_SIMPLE`}.

The debug instructions `load`, `save` and `random`, when used without the `--free_rw` option (see Section 5), account for cycles and power consumption for reading and writing through the crypto-processor read-write interface. However, the simulator doesn't account for any additional overheads due to the module that is interfacing with the crypto-core, e.g., a general-purpose micro-processor. Also, it doesn't account for the overheads of generating random values and encoding polynomials, which are typically performed outside the crypto-core in the general-purpose micro-processor. In summary, the simulator reflects only the behaviour of the crypto-core (read / write and computations) and not any external modules interfacing with it.

4 Example Code

In this section, example code snippets are provided for computations typically used in Ring-LWE and Module-LWE. Code can be annotated with Python-style single-line comments starting with "`#`". Polynomials in the NTT domain are written as $\hat{x}$, $\odot$ denotes polynomial coefficient-wise multiplication and $\star$ denotes polynomial multiplication in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. Several example programs are available in `https://github.com/banerjeeutsav/sapphire_sim/tree/master/programs`.

4.1 Ring-LWE

The following instructions calculate $b = a \star s + e$ for polynomials $a, s, e \in \mathcal{R}_q$. This is a typical computation in Ring-LWE and we use parameters $(n = 1024, q = 12289)$ similar to NewHope-1024 [5]. The initial random 256-bit seed `r0` is used to generate 256-bit *public-seed* (in `r0`) and 256-bit *noise-seed* (in `r1`) using SHA3-512.

```
# Parameter Configuration
config ( n = 1024 , q = 12289 )

# Random Seed Generation
random ( r0 )
sha3_init
sha3_512_absorb ( r0 )
r0 || r1 = sha3_512_digest

# Gen (a_hat)
rej_sample ( prng = SHAKE-128 , seed = r0 , c0 = 0 , c1 = 0 , poly = 0 )

# Sample (s)
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 0 , k = 8 , poly = 1 )

# Sample (e)
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 1 , k = 8 , poly = 2 )

# s_hat = NTT (s)
mult_psi ( poly = 1 )
transform ( mode = DIF_NTT , poly_dst = 5 , poly_src = 1 )

# e_hat = NTT (e)
mult_psi ( poly = 12 )
transform ( mode = DIF_NTT , poly_dst = 6 , poly_src = 2 )

# a_hat x s_hat
poly_op ( op = MUL , poly_dst = 0 , poly_src = 5 )

# b_hat = a_hat x s_hat + e_hat
poly_op ( op = ADD , poly_dst = 0 , poly_src = 6 )

# Save Data
save ( r0 , "data/pk_1.npy" )       # seed for a_hat
save ( poly = 0 , "data/pk_2.npy" ) # b_hat = a_hat x s_hat + e_hat
save ( poly = 5 , "data/sk.npy" )   # s_hat
```

Here $\hat{a}$, s, e are sampled in polynomials `(poly = 0)`, `(poly = 1)`, `(poly = 2)` respectively. Coefficients of s, e are sampled from binomial distribution with standard deviation $\sigma = \sqrt{k/2} = 2$. The polynomial $\hat{a}$ is considered to be in NTT domain. At the end, `(poly = 4)` contains the secret polynomial $\hat{s}$ and `(poly = 2)` contains the public polynomial $\hat{b} = \hat{a} \odot \hat{s} + \hat{e}$, both in NTT domain.

4.2 Module-LWE

The following instructions calculate $\boldsymbol{B} = \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}$ for matrix of polynomials $\boldsymbol{A} \in \mathcal{R}_q^{2\times 2}$ and vectors of polynomials $\boldsymbol{s}, \boldsymbol{e} \in \mathcal{R}_q^2$. The matrix $\boldsymbol{B}$ is calculated as:

$$\begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \end{pmatrix} = \begin{pmatrix} a_{00} \star s_0 + a_{01} \star s_1 + e_0 \\ a_{10} \star s_0 + a_{11} \star s_1 + e_1 \end{pmatrix}$$

This is a typical computation in Module-LWE and we use parameters $(n = 256, q = 7681)$ similar to Kyber-v1-512 [6]. The initial random 256-bit seed `r0` is used to generate 256-bit *public-seed* (in `r0`) and 256-bit *noise-seed* (in `r1`) using SHA3-512.

```
# Parameter Configuration
config ( n = 256 , q = 7681 )

# Random Seed Generation
random ( r0 )
sha3_init
sha3_512_absorb ( r0 )
r0 || r1 = sha3_512_digest

# Sample (S)
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 0 , k = 5 , poly = 4 )
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 1 , k = 5 , poly = 5 )

# S_hat = NTT (S)
mult_psi ( poly = 4 )
transform ( mode = DIF_NTT , poly_dst = 16 , poly_src = 4 )
mult_psi ( poly = 5 )
transform ( mode = DIF_NTT , poly_dst = 17 , poly_src = 5 )

# Gen (A_hat) - Row 0
rej_sample ( prng = SHAKE-128 , seed = r0 , c0 = 0 , c1 = 0 , poly = 0 )
rej_sample ( prng = SHAKE-128 , seed = r0 , c0 = 1 , c1 = 0 , poly = 1 )
# A_hat x S_hat - Row 0
poly_op ( op = MUL , poly_dst = 0 , poly_src = 16 )
poly_op ( op = MUL , poly_dst = 1 , poly_src = 17 )
init ( poly = 20 )
poly_op ( op = ADD , poly_dst = 20 , poly_src = 0 )
poly_op ( op = ADD , poly_dst = 20 , poly_src = 1 )

# Gen (A_hat) - Row 1
rej_sample ( prng = SHAKE-128 , seed = r0 , c0 = 0 , c1 = 1 , poly = 0 )
rej_sample ( prng = SHAKE-128 , seed = r0 , c0 = 1 , c1 = 1 , poly = 1 )
# A_hat x S_hat - Row 1
poly_op ( op = MUL , poly_dst = 0 , poly_src = 16 )
poly_op ( op = MUL , poly_dst = 1 , poly_src = 17 )
init ( poly = 21 )
poly_op ( op = ADD , poly_dst = 21 , poly_src = 0 )
poly_op ( op = ADD , poly_dst = 21 , poly_src = 1 )
```

```
# A * S = INTT (A_hat x S_hat)
transform ( mode = DIT_INTT , poly_dst = 8 , poly_src = 20 )
mult_psi_inv ( poly = 8 )
transform ( mode = DIT_INTT , poly_dst = 9 , poly_src = 21 )
mult_psi_inv ( poly = 9 )

# Sample (E)
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 2 , k = 5 , poly = 24 )
bin_sample ( prng = SHAKE-256 , seed = r1 , c0 = 0 , c1 = 3 , k = 5 , poly = 25 )

# B = A * S + E
poly_op ( op = ADD , poly_dst = 24 , poly_src = 8 )
poly_op ( op = ADD , poly_dst = 25 , poly_src = 9 )

# Save Data
save ( r0 , "data/pk_1.npy" )        # seed for A_hat
save ( poly = 24 , "data/pk_2.npy" ) # seed for b0 = a00 * s0 + a01 * s1 + e0
save ( poly = 25 , "data/pk_3.npy" ) # seed for b1 = a10 * s0 + a11 * s1 + e1
save ( poly = 16 , "data/sk_1.npy" ) # seed for s0_hat
save ( poly = 17 , "data/sk_2.npy" ) # seed for s1_hat
```

Here $\hat{a}_{00}$ / $\hat{a}_{10}$, $\hat{a}_{01}$ / $\hat{a}_{11}$, s_0, s_1, e_0, e_1 are sampled in polynomials `(poly = 0)`, `(poly = 1)`, `(poly = 4)`, `(poly = 5)`, `(poly = 24)`, `(poly = 25)` respectively. Coefficients of s_0, s_1, e_0, e_1 are sampled from binomial distribution with standard deviation $\sigma = \sqrt{k/2} = \sqrt{2.5}$. The polynomials s_0, s_1 are transformed using DIF-NTT, that is, with bit-reversed output, and the polynomials $\hat{a}_{00}$, $\hat{a}_{10}$, $\hat{a}_{01}$, $\hat{a}_{11}$ are considered to be in NTT domain. For the inverse operation, DIT-INTT is performed, which requires bit-reversed input, on the polynomials $\hat{a}_{00} \odot \hat{s}_0 + \hat{a}_{01} \odot \hat{s}_1$ and $\hat{a}_{10} \odot \hat{s}_0 + \hat{a}_{11} \odot \hat{s}_1$. At the end, `(poly = 16)`, `(poly = 17)` contain the secret polynomials $\hat{s}_0$, $\hat{s}_1$ in NTT domain and `(poly = 24)`, `(poly = 25)` contain the public polynomials $b_0 = a_{00} \star s_0 + a_{01} \star s_1 + e_0$, $b_1 = a_{10} \star s_0 + a_{11} \star s_1 + e_1$.

5 Running the Simulator

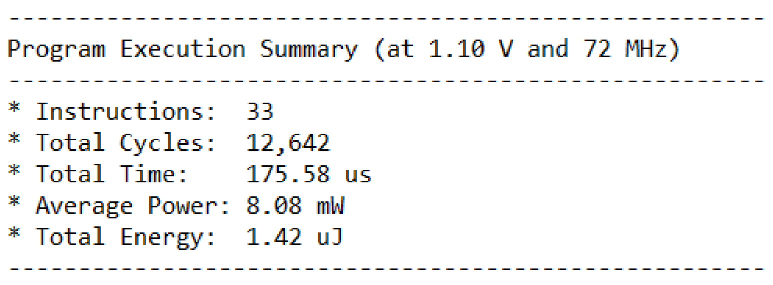
The Sapphire-Sim simulator can be run using Python (requires Python 3) as follows:

```
python sim.py --prog <program_file_path>
              --vdd <voltage>
              --fmhz <frequency_mhz>
              [ --verbose ]
              [ --free_rw ]
              [ --plot_power ]
              [ --cdt <cdt_file_path> ]
              [ --iter <num_iterations> ]
```

where `--prog`, `--vdd`, `--fmhz` are mandatory arguments providing the program file path, supply voltage ($\in [0.68, 1.21]$ V), operating frequency (in MHz) respectively. The simulator checks whether the operating frequency is below the maximum allowed frequency at specified supply voltage.

At the end of simulation, the following information are summarized (example shown in Fig. 2):

– Number of instructions executed (including branching)
– Total cycle count and execution time
– Average power consumption
– Total energy consumption



```
-----------------------------------------------------
Program Execution Summary (at 1.10 V and 72 MHz)
-----------------------------------------------------
* Instructions:  33
* Total Cycles:  12,642
* Total Time:    175.58 us
* Average Power: 8.08 mW
* Total Energy:  1.42 uJ
-----------------------------------------------------
```

Fig. 2. Example screen-shot of simulation summary.

The optional `--verbose` flag is used to enable or disable `print` instructions to display registers and polynomials. The optional `--free_rw` flag is used to enable or disable `load` / `save` / `random` instructions to skip cycle count and power consumption overheads associated with the crypto-processor's read-write interface. The optional `--plot_power` flag is used to enable or disable displaying the power consumption of the crypto-core as a function of time during program execution. Please note that this plot (example shown in Fig. 3) only provides a coarse estimate of the power consumption (only average power at the macro-op level) and is not at all intended (or suitable) for side-channel analysis.

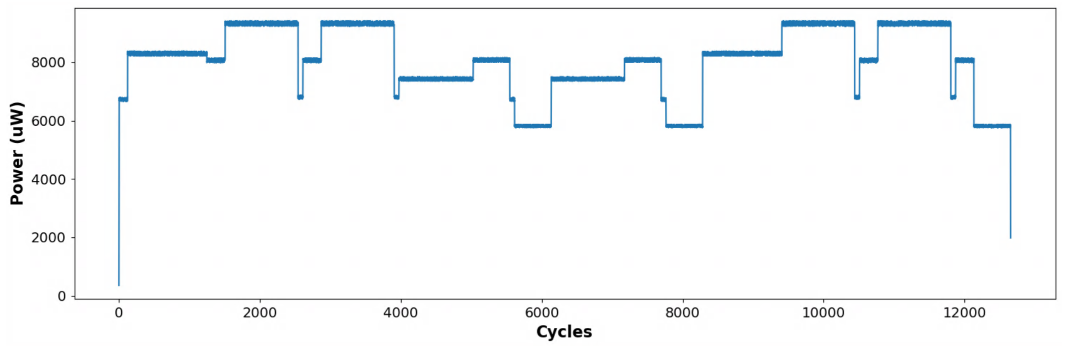


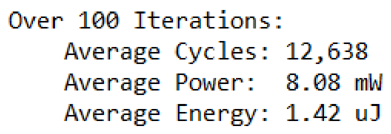
Fig. 3. Example screen-shot of simulated power consumption plot.

The optional `--cdt` flag is used to provide the CDT file path in case CDT-based sampling is used. The CDT file needs to be in plain text (for simplicity) with each CDT entry in separate lines. Sample discrete Gaussian CDT files are provided in https://github.com/banerjeeutsav/

sapphire_sim/tree/master/cdt_files. To help generate such CDT files for discrete Gaussian distributions with desired standard deviation `sigma`, CDT length `cdt_len` ≤ 64 and precision `prec` ≤ 32, a companion script gen_gaussian_cdt.py is also provided, which can be run as follows:

```
python gen_gaussian_cdt.py <sigma> <cdt_len> <prec> <out_cdt_file_path>
```

Finally, the `--iter` option can be used to indicate the number of iterations of program execution. When the specified number of iterations is greater than one, simulation summaries are reported for each iteration. At the end of all iterations, the average cycle count, power and energy consumption over all iterations are reported, as shown in Fig. 4. In case of more than one iteration, no power consumption plot is generated even if the `--plot_power` flag is set, and the simulator automatically prefixes `iter_<iter_count>_` to the names of all ".npy" data files generated during the simulation so that each iteration has its own separate set of data files and also it is easier to clean them up later. Note that specifying `--iter 1` is equivalent to not using the `--iter` option.



```
Over 100 Iterations:
    Average Cycles: 12,638
    Average Power:  8.08 mW
    Average Energy: 1.42 uJ
```

Fig. 4. Example screen-shot of simulation summary averaged over 100 iterations.

The simulator supports Verilog-style `` `define``, `` `ifdef`` and `` `endif`` macros to enable or disable chunks of code in the same program file. Nested `` `ifdef`` declarations are also allowed. This is useful when writing code for different steps, such as, KeyGen, Encrypt, Decrypt, in the same protocol. Although they are all written in the same file, the appropriate step can be enabled using its corresponding `` `define`` macro during simulation to get the power and performance numbers.

6 Simulation Results

The CPA-secure public key encryption (CPA-PKE) schemes NewHope [5], Kyber-v1 [6], R.EMBLEM [7] and LIMA [8] were simulated at 0.68 V and 12 MHz using Sapphire-Sim, and the results are shown in Table 1, as obtained from average over 100 iterations. The program files used for our simulations are provided in https://github.com/banerjeeutsav/sapphire_sim/tree/master/programs. Our implementations differ slightly from the reference software implementations of these schemes in some low-level details but otherwise follow the high-level algorithm specifications. Simulations were performed with the `--free_rw` and `--iter 100` flags with appropriate `` `define`` macro enabled. Sample ".npy" data files for each protocol are also provided in https://github.com/banerjeeutsav/sapphire_sim/tree/master/data. For R.EMBLEM, the error polynomials are sampled from the discrete Gaussian distribution with standard deviation $\sigma = 3.0$, support $s = 9$ and precision $r = 10$ bits. The corresponding CDT file, used with the `--cdt` flag, is provided in https://github.com/banerjeeutsav/sapphire_sim/blob/master/cdt_files/cdt_file_3p0_10_10.

The simulated cycle counts are very close to measured execution times in [4] after accounting for read/write overheads. Also, the simulated energy consumption is of the same order as the measured energy consumption reported in [9]. Finally, the `encode_compare` instruction in the decryption step results in `MATCH` for all cases, thus verifying correctness of the CPA-PKE implementations.

Table 1. Simulated performance of Ring/Module-LWE-based CPA-PKE at 0.68 V and 12 MHz

Protocol	Algorithm	Avg. Cycles	Avg. Power	Avg. Energy
NewHope-512	KeyGen	9,933	627.01 μW	518.96 nJ
	Encrypt	15,258	622.10 μW	790.96 nJ
	Decrypt	3,868	644.96 μW	207.89 nJ
NewHope-1024	KeyGen	21,076	628.91 μW	1.10 μJ
	Encrypt	32,535	623.29 μW	1.69 μJ
	Decrypt	8,481	644.02 μW	455.16 nJ
Kyber-v1-512	KeyGen	12,639	555.05 μW	584.59 nJ
	Encrypt	18,865	559.58 μW	879.69 nJ
	Decrypt	5,461	574.97 μW	261.66 nJ
Kyber-v1-768	KeyGen	22,007	547.27 μW	1.00 μJ
	Encrypt	30,116	553.48 μW	1.39 μJ
	Decrypt	7,344	575.14 μW	351.98 nJ
Kyber-v1-1024	KeyGen	33,467	541.25 μW	1.51 μJ
	Encrypt	43,459	548.17 μW	1.99 μJ
	Decrypt	9,227	575.24 μW	442.31 nJ
R.EMBLEM-512	KeyGen	16,382	487.29 μW	665.23 nJ
	Encrypt	30,716	482.23 μW	1.23 μJ
	Decrypt	6,700	739.58 μW	412.93 nJ
R.EMBLEM-1024	KeyGen	33,935	464.53 μW	1.31 μJ
	Encrypt	64,181	458.21 μW	2.45 μJ
	Decrypt	14,902	661.74 μW	821.77 nJ
LIMA-2p-1024	KeyGen	23,164	635.12 μW	1.23 μJ
	Encrypt	37,683	611.95 μW	1.92 μJ
	Decrypt	8,481	660.89 μW	467.08 nJ

7 Conclusion

A Python-based cycle-accurate simulator is presented for the Sapphire lattice-crypto processor which can be used to profile the performance of Ring-LWE and Module-LWE algorithms. This allows fast evaluation of lattice-based protocols with varying parameter choices but without any hardware design effort, which is especially important for a fast evolving field such as lattice-based cryptography. The simulator not only reports accurate cycle counts and execution times but also macro-operation-level power and average energy consumption modelled using measurements from the Sapphire test chip at various operating conditions. While several open-source tools are available for security estimation[2] and assembly-optimized software implementation[3] of Ring-LWE and Module-LWE protocols, this is the first attempt at quick simulation of hardware-accelerated implementations of the same. Our simulator is available at https://github.com/banerjeeutsav/sapphire_sim along with sample code for some CPA-PKE schemes. With some more effort, Sapphire-Sim can also be integrated with a simulator for the general-purpose micro-

[2] https://bitbucket.org/malb/lwe-estimator

[3] https://github.com/mupq/pqm4

processor in order to simulate CCA-KEM and signature schemes which require more complex data movement and control flow between the crypto core and the micro-processor. The Sapphire crypto core also supports power-of-two moduli which are currently not included in the simulator but may be added in the future in order to allow implementation of Ring-LWR and Module-LWR schemes.

References

1. C. Peikert, "A Decade of Lattice Cryptography," in *Now Publishers – Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 4, pp. 283-424, Mar. 2016.
2. L. Chen et al., "Report on Post-Quantum Cryptography," *NIST Technical Report*, no. 8105, Apr. 2016.
3. G. Alagic et al., "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process," *NIST Technical Report*, no. 8240, Jan. 2019.
4. U. Banerjee, T. S. Ukyab and A. P. Chandrakasan, "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols," in *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2019, no. 4, pp. 17-61, Aug. 2019.
5. T. Poppelmann et al., "NewHope – Algorithm Specifications and Supporting Documentation," *NIST Technical Report*, 2019. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions`.
6. P. Schwabe et al., "CRYSTALS-Kyber – Algorithm Specifications and Supporting Documentation," *NIST Technical Report*, 2019. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions`.
7. M. Seo et al., "EMBLEM and R.EMBLEM – Algorithm Specifications and Supporting Documentation," *NIST Technical Report*, 2018. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
8. N. P. Smart et al., "LIMA – Algorithm Specifications and Supporting Documentation," *NIST Technical Report*, 2018. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.
9. U. Banerjee, A. Pathak and A. P. Chandrakasan, "An Energy-Efficient Configurable Lattice Cryptography Processor for the Quantum-Secure Internet of Things," *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 46-48, Feb. 2019.