BIKE on Hardware

Jan Richter-Brockmann and Tim Güneysu

Ruhr-University Bochum

May 5, 2020

Contents

1 Hardware Implementation **2**
1.1 Subfunctions . . . 2
1.2 Implementation Results . . . 3
1.3 Estimations . . . 4
1.4 Optimizations . . . 4

1 Hardware Implementation

Since we decided to rely on BIKE-2 only, we also developed a new hardware implementation and concentrated our work mainly on the key generation because it contains a polynomial inversion which seems to be most challenging to implement. The realization is based on several submodules which will be presented first. Afterwards we provide some implementation results followed by estimations about further improvements.

1.1 Subfunctions

The key generation mainly consists of a sampler generating the private key, a uniform sampler generating σ, a module for the polynomial inversion and a multiplier.

All submodules are designed to work with Block-RAMs (BRAMs) storing all polynomials. This strategy reduces the amount of required registers which would otherwise be needed to hold intermediate results and keys. Especially for larger r, a solution without BRAM would not fit on smaller devices.

Multiplier Compared to the multiplier used in the reference implementation for the second round, the current multiplier was considerably improved regarding the throughput and latency. While the old multiplier requires $\lceil r/d \rceil \cdot r$ clock cycles to finish one polynomial multiplication, the new design finishes the multiplication within $\lceil r/d \rceil \cdot (\lceil r/d \rceil + 3)$ clock cycles where $d \in \{32, 64, 128\}$ and determines the bus width used internally. The implementation is similar to the polynomial multiplier proposed by Hu et al. [3], but follows a slightly different strategy so that a smaller latency is achieved[1] while it also requires less hardware resources.

Inversion As for the software implementation, the hardware module being responsible for the inversion of an element $a \in \mathcal{R}^*$ is also realized by Fermat's Little Theorem using

$$a^{-1} = a^{2^{r-1}-2}. \tag{1}$$

However, instead of following the recently proposed algorithm by Drucker at al. [1] to accomplish the inversion, the hardware implementation utilizes the algorithm proposed by Hu et al. [2, Alg. 1]. This algorithm is more advantageous for hardware when decompose all k-squarings into simple squarings as the total amount of required squarings is decreased. Besides $r-2$ squarings, the algorithm requires $\lfloor \log(r-2) \rfloor + |supp(r-2)|$ multiplications.

Squaring For the polynomial squaring we developed a module which performs one squaring within $\lceil r/32 \rceil$ clock cycles. This procedure allows us to write one entire part of 32 bits of the destination polynomial in one clock cycle after a short initial phase.

[1]The design by Hu et al. requires $\lceil r/d \rceil^2 + 18 \lceil r/d \rceil - 9$ clock cycles.

Table 1: Implementation results on an Artix-7 (xc7a100) FPGA for the key generation.

	Resources				Performance		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles (average)	MHz	ms
Level 1	1 865	589	4	590	7 370 429	135	54.54
Level 3	1 884	557	5	593	30 447 947	131	231.4

Sampler In order to implement the key generation, two samplers are required which can generate polynomials with r bits and a Hamming weight of $w/2$ and an uniform vector with ℓ bits representing σ.

In general, the first sampler is realized as rejection sampler such that a sampling of a vector v of length l requires a $\lceil \log(l) \rceil$ bit random input r_{samp}. In case $r_{\text{samp}} \geq l$ the randomness will not be used avoiding a biased results. However, every sampling of a single bit takes two clock cycles. In the first clock cycle the current value of the vector v is read depending on the current randomness. If the target bit is already set, the value is written back unaltered in the next clock cycle. If the target bit is zero, it can be set to one and afterwards written back to the memory.

The second sampler applied to generate uniform vectors requires a random input of 32 bit which can be directly used to set the bits of the target vector.

1.2 Implementation Results

In the following we describe the composition of the above characterized modules to implement the key generation and provide implementation results including hardware utilization and timing evaluations.

On the top level, the key generation in hardware consists of four modules: the sampler for the private key, the sampler for σ, a BRAM storing both keys, and the inversion module which also performs the multiplication of h_1 times h_0^{-1} as there is already a multiplier included. For the sampler we decided to fix the data width d to 32 bits as only one bit can be sampled at a time and increasing d would just result in a larger area footprint and no increased throughput. The BRAM module gets its input data from the sampler and is only read by the inversion module. Besides the squaring module and the multiplier, the inversion unit contains three BRAMs storing intermediate results. Finally, the public key is stored in one of them and returned to the output.

Table 1 provides implementation results about the utilized hardware resources and the timing behavior for both security levels generated for an Artix-7 (xc7a100) FPGA. The implementation for Level-1 requires roughly 7.3 million clock cycles to finish the key generation but only consumes 590 slices and is therefore perfectly suited for smaller devices.

1.3 Estimations

In this section, we shortly discuss a hardware implementation for the encapsulation which is currently under development. The encapsulation consists of different subfunctions: a uniform sampler to generate the message m, a multiplier to compute c_0, and the three random oracles $\mathbf{H}, \mathbf{K}, \mathbf{L}$. The random oracles are based on a SHA-384 which can be compared to a SHA3-384[2] regarding the implementation costs. However, the realization of $\mathbf{H}$ is some more challenging and costly on hardware as it contains an additional AES core. Nevertheless, for the message generation the sampler from the key generation can be reused. Also the multiplier, which was highly optimized, can be reused and speeds up the encapsulation notably.

Currently the hardware implementation of the encapsulation is not completely finished, but we will make it public on our website as soon as possible. Despite, we would like to provide some first estimations regarding the latency of the encapsulation for both security levels and for different d in Table 2.

Table 2: Estimated latency in clock cycles for the encapsulation for both security levels and for different d.

Security	$d = 32$	$d = 64$	$d = 128$
Level 1	162 000	50 000	22 000
Level 3	610 000	164 000	52 000

1.4 Optimizations

To improve the latency of the hardware implementation of the key generation, we identify different strategies which should be discussed in this section. As already implied in subsection 1.1, the whole implementation for the key generation can be made scalable such that all submodules would be compatible with the parameter $d \in \{32, 64, 128\}$.

Other optimizations address the implementation of the inversion module as it is the main part of the key generation. The first strategy includes a second k-squaring module designed for a fixed k and being able to perform the k-squaring in the same time as a simple squaring module would do. The second strategy would also use an additional squaring module which would be able to perform an arbitrary k-squaring in r clock cycles.

Applying those optimization strategies, we expect the improvements depicted in Figure 1. First, using the above described approach implementing the key generation, the latency can be reduced from 7.3 million clock cycles to roughly 1.5 million clock cycles just by scaling the parameter d. By applying this optimization, we achieve a decreased latency but expect a eight times higher hardware utilization. Second, implementing optimization strategy I would further decrease the latency by more than half of the clock

[2]A SHA3-384 module was used in the hardware implementation of Round-2.

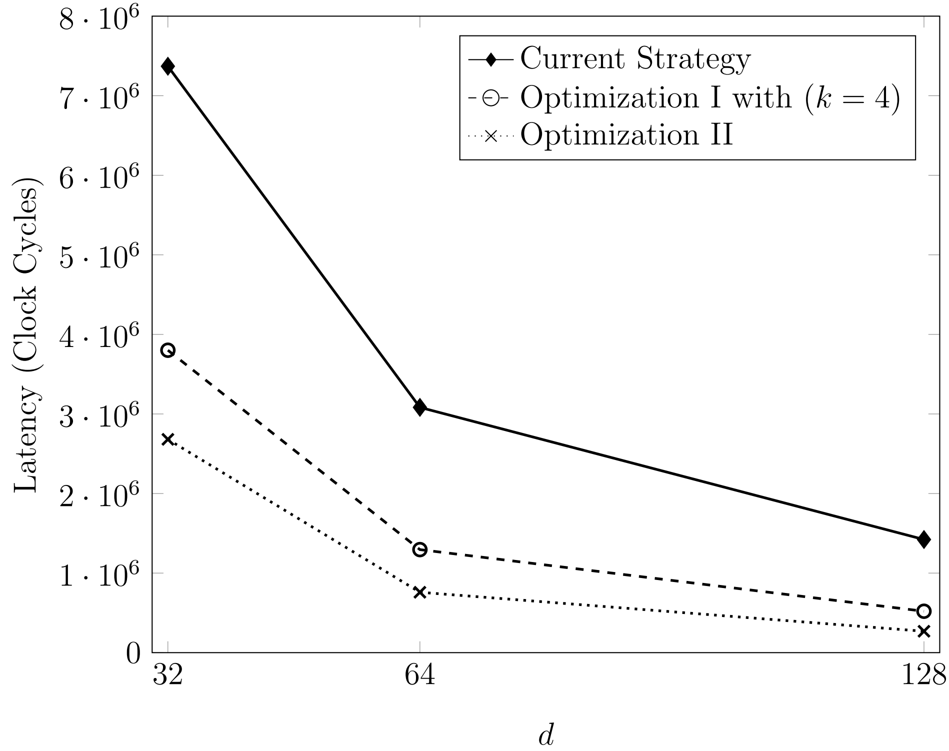

Figure 1: Estimated latencies for different optimization strategies for $r = 12\,323$.

cycles. However, implementing a second k-squaring module with $k = 4$ would probably increase the hardware utilization disproportional and would eventually lead to exploding implementation costs. Third, optimization strategy II further improves the latency remarkable. We expect reasonable implementation costs such that this strategy probably provides the best trade-off.

References

[1] Nir Drucker, Shay Gueron, and Dusan Kostic. *Fast polynomial inversion for post quantum QC-MDPC cryptography*. Cryptology ePrint Archive, Report 2020/298. https://eprint.iacr.org/2020/298. 2020.

[2] Jingwei Hu et al. "Fast and Generic Inversion Architectures Over $GF(2^m)$ Using Modified Itoh–Tsujii Algorithms". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 62.4 (2015), pp. 367–371.

[3] Jingwei Hu et al. "Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKE". In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2019, pp. 231–234.