Efficient FPGA Implementations of LowMC and Picnic

Daniel Kales[1], Sebastian Ramacher[2], Christian Rechberger[1], Roman Walch[1,3], and Mario Werner[1]

[1] Graz University of Technology, Graz, Austria
{daniel.kales,christian.rechberger,roman.walch,mario.werner}@iaik.tugraz.at
[2] AIT Austrian Institute of Technology, Vienna, Austria
sebastian.ramacher@ait.ac.at
[3] Know-Center GmbH, Graz, Austria

Abstract. Post-quantum cryptography has received increased attention in recent years, in particular, due to the standardization effort by NIST. One of the second-round candidates in the NIST post-quantum standardization project is Picnic, a post-quantum secure signature scheme based on efficient zero-knowledge proofs of knowledge. In this work, we present the first FPGA implementation of Picnic. We show how to efficiently calculate LowMC, the block cipher used as a one-way function in Picnic, in hardware despite the large number of constants needed during computation. We then combine our LowMC implementation and efficient instantiations of Keccak to build the full Picnic algorithm. Additionally, we conform to recently proposed hardware interfaces for post-quantum schemes to enable easier comparisons with other designs. We provide evaluations of our Picnic implementation for both, the standalone design and a version wrapped with a PCIe interface, and compare them to the state-of-the-art software implementations of Picnic and similar hardware designs. Concretely, signing messages on our FPGA takes 0.25 ms for the L1 security level and 1.24 ms for the L5 security level, beating existing optimized software implementations by a factor of 4.

Keywords: LowMC · FPGA · digital signatures · NIST PQC · Picnic

1 Introduction

Cryptographic primitives with low multiplicative complexity have many interesting applications in higher-level protocols.[4] Recently, the post-quantum secure digital signature scheme Picnic [20,19] used zero-knowledge proof of knowledge schemes to build a signature based on the knowledge of a pre-image of a one-way function (OWF). Since the size of the proof of knowledge is directly related to the

[4] For example, they find use as pseudo-random functions (PRF) in secure multiparty computation (MPC) [5,31,45] to handle encrypted data, as oblivious PRFs in private set intersection (PSI) [28,37], but also enable the elimination of ciphertext expansion in homomorphic encryption schemes [42,5].

number of AND gates in the OWF, Picnic employed block ciphers with low multiplicative complexity. In particular, the designers of Picnic chose LowMC [5], a very parameterizable block cipher design, to build the OWF. Picnic is currently a round 2 candidate in the NIST post-quantum cryptography project [1] and, since the construction lends itself to design more complex statements, it has also been extended to other signature variants including ring and EPID signatures [22,38,14] as well as double-authentication preventing signatures [21]. All of those signatures only rely on symmetric-key primitives for their security guarantees.

LowMC allows its users to select parameters suitable for the intended application. For the use in Picnic, this means that one can select instances with a reduced data complexity and thereby reducing the number of required rounds. More importantly, it can also be parametrized in such a way that the number of multiplication gates – in this case AND gates – is minimized. For classical security of 128 bits, LowMC only requires 861 AND gates for full security and 546 AND gates in the reduced data complexity case. In comparison to that, other candidates for lightweight cipher designs require significantly more AND gates to achieve full data security: Simon requires 4352 AND gates [10], Kreyvium requires 1537 AND gates [17], and Fantomas requires 2112 AND gates [32] (cf. [20, Section 6.1] and [3, Table 1]). None of them come close to the numbers of LowMC. Only recently, GMiMC [2] was proposed, which can be reduced to 783 AND gates and can compete in the low data complexity scenario. For reference, AES-128 implemented over GF(2) requires more than 5000 AND gates [16] and a round reduced version for the low data complexity case would amount to 3200 AND gates (including key schedule) [15].

While the choice of LowMC with small multiplicative complexity significantly reduces the signature size of Picnic, the number of LowMC rounds has to be increased for security. Conversely, with higher multiplicative complexity, fewer rounds are required to achieve a secure design. Since each round consists of a matrix multiplication involving the full state, the number of rounds essentially define the runtime characteristics of Picnic. Additionally, one matrix is sampled uniformly at random for each round during instance generation. In practice, this means that the size of the constants stored in implementations also grows linearly in the number of rounds. For the instances selected for NIST's L5 security level of 128-bit post-quantum security, the constants sum up to 621 KB. Recent optimizations of the linear layer by Dinur et al. [23] reduced the storage requirements for the constants down to 129 KB. Even with these optimizations, the sheer size of involved matrices seems to prohibit an implementation on resource-constrained devices, like microcontrollers or FPGAs. While the size of constants is less of a problem for software implementations running on desktops, servers or mobile phones, the size of these constants has a direct impact on the area of a hardware design or the hardware utilization of FPGA designs, respectively.

As the NIST post-quantum project progressed into the second round, the performance of the candidates is becoming a more important criterion. Consequently, NIST published targets for optimized implementations. Since optimized

software implementations of LowMC and Picnic already exist, we focus on the implementation of several variants on FPGA platforms, including the Xilinx Kintex-7 as well as the Xilinx Artix-7. The latter is one of the optimization platforms recommended by NIST.

1.1 Contribution

Our contribution can be summarized as follows. We provide the first FPGA implementation of LowMC using a state machine design. Due to the structure of LowMC, the evaluation of the encryption algorithm requires a large number of constants in the form of uniformly random matrices and vectors. We adapt the recent result of Dinur et al. [23] to our FPGA implementation and are thereby able to significantly reduce the hardware utilization of our design compared to a naïve implementation.

We combine the LowMC implementation and custom Keccak modules to instantiate the complete Picnic design on a Xilinx Kintex-7 board. This implementation conforms to the round 2 submission of the Picnic signature scheme [19] and supports the L1 and L5 parameter sets. Additionally, we port our implementation of Picnic-L1 to the Xilinx Artix-7 board. The implementation is flexible enough to support signing only and verification only versions besides the full version without significant overhead. However, our implementation focuses on the Fiat-Shamir transformed version of Picnic due to recent results [24,18] improving the confidence in its post-quantum security. Furthermore, the implementation also conforms to the proposed guidelines for hardware designs [26] of post-quantum schemes in the NIST standardization project to facilitate easier comparisons with other designs.

We evaluate the performance of our FPGA design and provide comparisons to optimized software implementations of Picnic. Our design performs up to a factor 4 faster for signing and up to a factor 3 faster for verification than SIMD optimized software implementations. We also compare our design to other hardware designs of signature schemes including SPHINCS [11]. While the Picnic design has a higher hardware utilization, it performs significantly better in run time than the SPHINCS design.

We discuss potential modifications to the LowMC cipher design that would benefit the FPGA implementation of Picnic. In particular, the suggested changes improve the hardware utilization by up to 30% and makes it possible to fit a Picnic implementation for the 128-bit post-quantum security level on the NIST recommended Artix-7 evaluation board.

Finally, we also provide a pipeline design of LowMC for high throughput scenarios. This design may also be of interest in other contexts such as PSI protocols [37], or fast database joins on secret shared data [40].

1.2 Related Work

Efficient hardware implementations are a very active point of research, and the NIST post-quantum standardization project only amplifies this [1]. In the fol-

lowing, we discuss other works in this area, with a focus on hardware implementations of post-quantum algorithms. We still want to mention that for a wide variety of primitives and schemes, hardware implementations have been proposed over the years [41,6,48,47], among others.

Basu et al. [9] provide an evaluation of 11 of the second-round candidates of the NIST post-quantum standardization project. Their approach is based on an automated synthesis of post-quantum accelerators based on the C code provided by the submissions to NIST. They analyze the designs for both FPGA and ASIC targets and compare their runtime and hardware utilization. But since Picnic is a fairly complex design with a large number of individual primitives and constants, an FPGA implementation is not straightforward and can not be synthesized easily from the software implementation. Therefore, Basu et al. do not include Picnic in their evaluation.

Besides this generic approach, Amiet et al. [7] presented an FPGA accelerator for SPHINCS-256 [11], a predecessor of SPHINCS+ [12], another candidate in the NIST post-quantum standardization project. Their implementation provides a highly optimized ChaCha12 pipeline, which is the core of the SPHINCS design. They report signing times of 1.53 ms on the same FPGA we target in our work.

Wang et al. [50] presented FPGA implementations of the Niederreiter cryptosystem using binary Goppa codes. For key encapsulation mechanisms and key exchange, Howe et al. [34] presented implementations of FrodoKEM, a post-quantum key encapsulation mechanism, for FPGAs and microcontrollers. Roy et al. [46] give an optimized implementation of SIKE, a post-quantum key exchange algorithm based on supersingular isogenies, for FPGA platforms. In a different approach, Albrecht et al. [4] repurpose existing RSA coprocessors to speed up computations of RLWE-based schemes, concretely for KEM Kyber.

2 Preliminaries

We will give an overview of LowMC, Picnic and its main building blocks.

2.1 LowMC

LowMC [5] is a block cipher designed to reduce the number of AND gates needed for symmetric encryption. The design of the cipher is based on the substitution-permutation network (SPN) design strategy, with the choice to move to a partial substitution layer instead of applying the Sbox on the full state. The parameters block size (n), key size (k), number of Sboxes per round (m), allowed data complexity (d) and the number of rounds (r) are parameterizable according to the LowMC v3 round formula [44]. This formula calculates the lowest number of rounds necessary to provide secure encryption for the given parameter set. The script for determining the number of rounds for a given set of LowMC parameters can be found in the official GitHub repository [39].

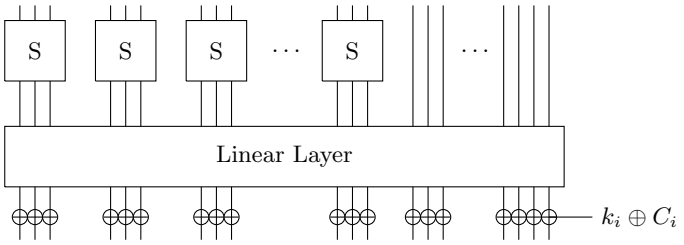


Fig. 1: One round of encryption with LowMC (modified from [5]).

A LowMC encryption starts with an initial whitening by XORing the first round key to the plaintext, followed by r rounds. As depicted in Figure 1, one round consists of four steps: (i) SboxLayer, (ii) LinearLayer, (iii) ConstantAddition and (iv) KeyAddition. In the SboxLayer, m 3-bit Sboxes are applied to the first $s = 3 \cdot m$ bits of the state. The remaining bits of the state are not affected by the SboxLayer. The Sbox is defined as

$$S(a, b, c) = (a \oplus b \cdot c, a \oplus b \oplus a \cdot c, a \oplus b \oplus c \oplus a \cdot b)\,,$$

with three GF(2) inputs and outputs. From this definition, it is obvious that only 3 AND gates are required per Sbox. In the LinearLayer, the state is multiplied with a pseudorandomly generated matrix $L_r \in \mathrm{GF}(2)^{n \times n}$, where r is the current round. The matrices are chosen pseudorandomly from the set of all invertible binary $n \times n$ matrices during the instantiation of LowMC. During the ConstantAddition the vector $C_r \in \mathrm{GF}(2)^n$ is XORed to the state, where r describes the current round. The vectors are chosen pseudorandomly during the instantiation of LowMC. During KeyAddition, the round key of the current round is XORed to the state. All round keys are generated as a result of the multiplication of the master key with the matrix $K_r \in \mathrm{GF}(2)^{n \times k}$, where r is the current round. The matrices are chosen pseudorandomly from the set of all full-rank binary $n \times k$ matrices during the instantiation of LowMC.

2.2 Picnic and ZKB++

The Picnic signature scheme is based on zero-knowledge proofs of knowledge of pre-images of one-way functions. Currently, Picnic supports two proof systems: ZKB++ [20] and KKW [38]. They both improve on the "MPC-in-the-head" paradigm [36], which describes a generic way to turn MPC protocols into zero-knowledge proofs. In this work we focus on the parameter set using ZKB++, since it is more efficient in runtime.

ZKB++ builds the zero-knowledge proof system from $(2, 3)$-circuit decomposition, which we describe in more detail. Let ϕ be some circuit and $y = \phi(x)$, where x is some secret input and y is the publicy-known output. Within $(2, 3)$-circuit decomposition, the computation is decomposed in the following way [29]:

- Share splits the input into three input shares.

– UPDATE advances the computation one gate at a time, computes the wire values for the next gate and returns the updated view.
– OUTPUT produces the output shares based on the final view.
– RECONSTRUCT recomputes the output from the three output shares.

The decomposition of the circuit has to satisfy correctness and 2-privacy [29]:

Correctness: The reconstruction of the output shares y_i must always be the result of the original relation $y = \phi(x)$.
2-Privacy: It should not be possible to reveal information about the private key x by publishing any information on any two players.

In ZKB++, $(2,3)$-circuit decomposition is constructed as follows: Let R be an arbitrary finite ring and ϕ a function such that $\phi\colon R^m \to R^\ell$ can be expressed by an n-gate arithmetic circuit over the ring using addition (respectively multiplications) by constants, and binary addition and binary multiplication gates. A $(2,3)$-decomposition of ϕ is then given by:

$\mathsf{Share}(x, k_1, k_2, k_3)$: Samples random $x_1, x_2 \in R^m$ from k_1 and k_2 and computes x_3 such that $x_1 + x_2 + x_3 = x$. Returns views containing x_1, x_2, x_3.
$\mathsf{Update}_i^{(j)}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1})$: Computes player P_i's view of the output wire of gate g_j and appends it to the view. For the k-th wire w_k where $w_k^{(i)}$ denotes P_i's view, the update operation is defined as follows:
Addition by constant ($w_b = w_a + c$): $w_b^{(i)} = w_a^{(i)} + c$ if $i = 1$ and $w_b^{(i)} = w_a^{(i)}$ otherwise.
Multiplication by constant ($w_b = c \cdot w_a$): $w_b^{(i)} = c \cdot w_a^{(i)}$
Binary addition ($w_c = w_a + w_b$): $w_c^{(i)} = w_a^{(i)} + w_b^{(i)}$
Binary multiplication ($w_c = w_a \cdot w_b$): $w_c^{(i)} = w_a^{(i)} \cdot w_b^{(i)} + w_a^{(i+1)} \cdot w_b^{(i)} + w_a^{(i)} \cdot w_b^{(i+1)} + R_i(c) - R_{i+1}(c)$ where $R_i(c)$ is the c-th output of a pseudo-random generator seeded with k_i.
$\mathsf{Output}_i(\mathsf{view}_i^{(n)})$: Return the ℓ output wires stored in the view $\mathsf{view}_i^{(n)}$.
$\mathsf{Reconstruct}(y_1, y_2, y_3)$: Computes $y = y_1 + y_2 + y_3$ and returns y.

Note that the player P_i can compute all gate types except for binary multiplication gates locally as the latter requires inputs from P_{i+1}. In other words, only outputs of binary multiplication gates need to be serialized as part of the communication transcript, and thus the view size and consequentially the signature size of PICNIC depend on the size of the ring R and the number of these gates.

To create a proof the prover repeats the $(2,3)$-circuit decomposition protocol T times. For each run, the prover commits to the view of each player P_i consisting of the input share, a communication transcript, and the output share. After all T runs, the prover sends all the output shares and commitments for each run and player to the verifier, who responds with a challenge vector c. The challenge tells the prover which two of the three players should be corrupted per run and therefore which views should be published as part of the proof. Since the decomposition satisfies the 2-privacy property, no information is leaked on the

secret key by publishing the views of two players. The verifier then recalculates the two opened views and checks, (1) whether the opened views were calculated correctly, (2) and if the three output shares can be reconstructed to y.

Each run gives some assurance that the prover knows the secret key x, therefore increasing the number of runs T decreases the probability that the prover can cheat without the verifier catching him at least once. Due to the nature of the circuit decomposition, the prover could potentially cheat in 2 of the 3 possible challenges per run; therefore we calculate the probability for him to cheat without getting caught as $(2/3)^T$.

ZKB++ is a Σ-protocol, i.e., an interactive proof, that can be made non-interactive by applying standard techniques such as the Fiat-Shamir (FS) transformation [27]. FS transformed Σ-protocols compute the challenge c as the output of a random oracle on the first message from the prover to the verifier, which contains the commitments to the shares of the circuit evaluation. This results in a non-interactive zero-knowledge proof protocol secure in the random oracle model. To obtain a signature scheme the message is included in the call to the random oracle as well.

In Picnic, the circuit used for the circuit decomposition is $C = \text{LowMC}_k(p)$, where k is a LowMC secret key and (p, C), a corresponding plain-/ciphertext pair, which is known publicly and constitutes the public key. A signature is then a proof of knowledge of a k satisfying this relation.

2.3 Picnic Instances and Parameters

For each of the three security levels $S \in \{128, 192, 256\}$, there exist two variants of the Picnic algorithms differing in the choice of transformations turning the Σ-protocol into a signature scheme: one variant is based on the Fiat-Shamir (FS) transformation, and the other is based on the Unruh (UR) transformation [49]. In contrary to the FS transformation, which makes the resulting non-interactive Σ-protocol secure in the random oracle model, the UR transformation is provably secure in the quantum random oracle model (QROM) [13], where an adversary can query the random oracle in quantum superposition. However, recent results by Don et al. [24] and Chailloux [18] show that the specific use of the FS transformation in Picnic is also secure in the quantum random oracle model. Therefore, we focus our implementations on the variants of Picnic using the FS transformation and ignore the more costly variants based on the Unruh transformation, that would require additional Keccak instances to be fitted into the FPGA design.

Picnic uses LowMC to reduce the size of the overall proof and thus the signature. The proof contains a transcript of a party, i.e., the view of the party for each AND gate in the circuit. Due to the additive secret-sharing used, XOR gates can be computed locally and do not influence the signature size. Therefore, LowMC is used instead of other lightweight ciphers, since, as discussed in Section 1, alternatives to LowMC require significantly more AND gates. Table 1 shows the LowMC instances that are used in Picnic and their AND gate counts. We note that those instances are selected to provide a trade-off between

Table 1: Picnic parameters with LowMC instances (block size n, key size k, # of Sboxes m, rounds r), sizes of public key pk, secret key sk and signatures σ.

			LowMC				Hash/KDF		Sizes		
Parameter Set	S	T	n	k	m	r	Algorithm	ℓ	pk	sk	σ
Picnic-L1-FS	128	219	128	128	10	20	SHAKE128	256	32	16	≤ 34032
Picnic-L3-FS	192	329	192	192	10	30	SHAKE256	384	48	24	≤ 76772
Picnic-L5-FS	256	438	256	256	10	38	SHAKE256	512	64	32	≤ 132856

signature size and runtime [20]. We want to note though, that even when the instances are selected based on this trade-off, they can still be represented with a lower number of AND gates than alternative cipher designs.

Table 1 shows the parameters of the different Picnic versions based on the Fiat-Shamir transformation. The expected security of the various instances corresponds to S bits against classical attacks and $S/2$ bits against quantum attacks. The parameter T describes the number of repetitions of ZKB++ required to reduce the soundness error to the desired security level [19]. Additionally, Table 1 shows the different key and signature sizes for the Picnic instances. One particular optimization of ZKB++ [20] has the result that the signature size is dependent on the challenge, because for one of the players in the MPC protocol we need to include some auxiliary information in the proof. Therefore, the expected signature size will be smaller than values specified in the table.

In the remainder of this work, we focus our implementation on the Picnic instances with security levels $S \in \{128, 256\}$ based on the FS transformation, namely Picnic-L1-FS and Picnic-L5-FS. Similarly, for LowMC the focus lies on instances with 128 and 256-bit block and key sizes.

Picnic2 *Instances.* In the second round of the NIST post-quantum standardization project, the Picnic team introduced an additional new parameter set, called Picnic2. The main difference in the new parameter set is the choice of the underlying proof system. In Picnic2, ZKB++ was complemented with KKW [38], which is also based on the "MPC-in-the-head" paradigm, but uses a different MPC protocol with precomputation. The nature of the new proof system allows for shorter signatures, but it has an increased number of players in the simulated MPC protocol resulting in longer signing and verification times when compared to Picnic. An evaluation of these additional parameter sets on FPGA platforms is an interesting topic for future work.

3 Implementation

We will now describe our implementations[5] of LowMC and the Picnic signature algorithm on an FPGA platform. We first give insight into the design of

[5] All implementations are available at `https://github.com/IAIK/Picnic-FPGA`.

the main module of Picnic, the computation of LowMC. Following that, we show how to combine this module with several SHAKE modules to instantiate the full Picnic signature scheme for the L1 and L5 security levels. In our implementation, we use the dual-port RAM module available as in the open-source framework MEMSEC [52,51].

Target Platform. For our design, we target a Xilinx Kintex-7 board – concretely we use a Xilinx Kintex-7 FPGA KC705 Evaluation Kit. Our target FPGA has 203800 lookup-tables (LUTs), 407600 flip-flops (FF) and 445 BRAMs available. In an announcement on the official mailing list of the NIST post-quantum standardization project, it was specified that implementors should target the Artix-7 platform due to its widespread use. Since the toolchain we use, Xilinx Vivado, also supports Artix-7 platforms as a target, adaptation to this platform is straightforward, and we discuss the resulting Artix-7 resource utilization in Section 4.1.

3.1 Optimized VHDL Implementation of LowMC

One of the major modules in our Picnic design is the evaluation of the LowMC block cipher. During Picnic's signing process, a proof of knowledge is generated by evaluating the LowMC encryption function in an MPC protocol. As discussed in Section 2.2, this is done by applying the $(2, 3)$-circuit decomposition as defined by ZKB++ to the LowMC circuit. In terms of the matrix multiplications, the sharing of the circuit requires the 3-fold evaluation for signing.

In this section, we discuss our design choices and the difficulty of a LowMC VDHL implementation and compare a straightforward standalone implementation of LowMC with a standalone implementation using the optimizations by Dinur et al. [23]. We shortly give an intuition of these optimizations in the following Sections and refer the reader to [23] for a more detailed explanation.

Design Choices. The difficulty in implementing LowMC (and consequently Picnic) in VHDL arises from the high number of constants involved in the matrix multiplications in LowMC's linear layer and round key schedule. For the LowMC instance in Picnic-L5, 621 kB of constants are required which can be reduced to 129 kB by using the optimizations in [23]. Usually, we consider using block RAM (BRAM), RAM cells directly located on FPGAs, for storing a large amount of constants. The Kintex-7 FPGA comes with dual-port BRAM cells with a capacity of 36 kB each, which are capable of providing at most 72 bits during one clock cycle at each port. During one round we multiply the inner state of LowMC to an $S \times S$ bit (256×256 for Picnic-L5) matrix. Considering a high-performance implementation, where we want to perform the matrix multiplication in one clock cycle, we would have to use ≈ 455 BRAM cells in parallel, which exceeds the number of available cells. The alternative multi-cycle approach would, therefore, necessarily lower the performance of the implementation. Furthermore, in the case of high BRAM cell usage, additional clock frequency penalties have to be expected due to increasing routing delays.

In our implementation, we decided to encode the constants for the matrices in lookup-tables (LUTs). This decision implies a high hardware utilization of our design, but comes with the advantage of fast matrix multiplications (1 clock cycle each) and therefore with the best performance. A low area implementation of LowMC (and consequently Picnic) using BRAMs instead of LUTs for constants could be an interesting topic for future work.

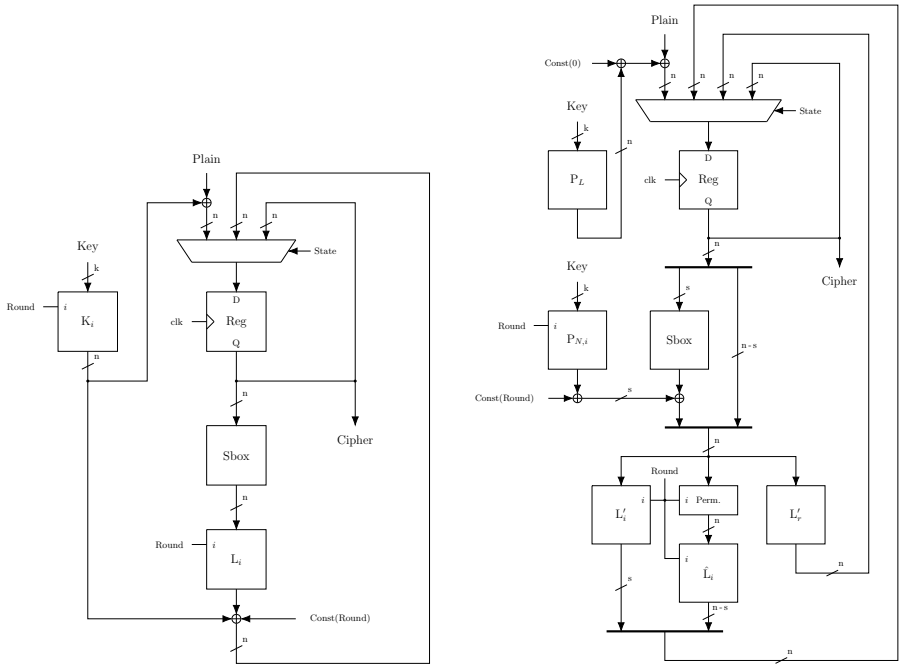
LowMC *Optimizations.* The main idea behind the optimizations proposed in [23] is that all operations except the Sbox layer are linear. Furthermore, only part of the state is affected by the Sbox layer. For example, consider the key addition: we can swap the order of the linear layer and the key addition by multiplying the round key with the inverse of the linear layer. Subsequently, we can move part of the key addition through the identity part of the Sbox layer and combine this part of the key addition with the key addition of the previous round. This process can be repeated recursively until we have combined a large part of the key additions in the initial key addition before the first round. The same process can be repeated for the round constants. Using some more advanced linear algebra properties, we can also move parts of the linear layer matrix multiplication to the next or previous round, again repeating this process to combine parts of the linear layer in the last or first round.

Figure 2a shows the VHDL design of LowMC without the optimizations. In Figure 2b, we present the design with the optimizations applied. Without the optimizations, there is only one implementation for all the rounds, and the matrix multiplications affect the entire state. In the optimized implementation, there are 5 different matrix multiplication modules, each for a matrix with different dimensions. The Sbox layer, round key, and constants of the new implementation only affect the first s bits of the state, and the linear layer matrix multiplication follows the algorithm in [23].

Optimized Hardware Utilization. The impact of the optimizations depends on the concrete LowMC instance. It especially depends on the number of Sboxes m and the resulting size of the non-linear layer. The fewer Sboxes per round, the more significant is the effect of the optimizations. The concrete effect of the optimizations can be seen in Table 2, where the required lookup-tables (LUTs) of the LowMC VHDL implementation are shown for the two different LowMC instances used in Picnic-L1-FS and Picnic-L5-FS. The instance for security level L1 only requires about a third of the LUTs required before, and the instance for security level L5 only requires about a fifth of the LUTs of the straightforward version. Without the optimizations, it would not even be possible to synthesize one LowMC instance for security level L5 on our FPGA board, whereas we require several instances for the Picnic implementation. The improvement for LowMC instances with larger non-linear layers is smaller, though.

3.2 Pipeline versus State Machine

Besides the implementation of LowMC using a simple state machine, we also provide an alternative implementation using a pipelined design. While both de-



(a) LowMC implementation without optimizations of the round key and linear layer computations.

(b) LowMC implementation with optimized round key computation and linear layer evaluation.

Fig. 2: State diagrams of different LowMC implementations.

Table 2: LUTs of one LowMC with/without optimizations (203800 available).

LowMC instance	LowMC n	k	m	r	without opt. LUTs	% LUTs	with opt. LUTs	% LUTs	Improv. %
Picnic-L1-FS	128	128	10	20	42395	20.80 %	13558	6.65 %	68.02 %
Picnic-L5-FS	256	256	10	38	209348	102.72 %	44431	21.8 %	78.78 %

signs have a latency of r cycles to get a specific ciphertext, the pipelined design has a much higher throughput with 1 ciphertext per cycle. The state machine design, on the other hand, has to wait for an encryption to be finished before it can process another plaintext and therefore has a throughput of 1 ciphertext per r rounds. However, the state machine design requires fewer lookup tables on an FPGA, because the LowMC round only needs to be instantiated once. For the Picnic coprocessor, we use the state machine design due to smaller hardware utilization. When interested in higher throughput, for example, when it is used as an oblivious pseudo-random function in a PSI protocol, the pipeline design is the better choice.

3.3 Optimized VHDL Implementation of Picnic

We now use the LowMC implementation as a building block for our Picnic coprocessors. In the following, we shortly describe the other different submodules and finally, the high-level design of the Picnic coprocessors.

LowMC-*MPC.* In Picnic, three copies of the LowMC encryption circuit are evaluated with three random additive shares of the secret key. Since the secret-sharing used is additive, XOR gates can be computed locally for each part, while some communication between the parties and randomness is required for computing an AND gate. While a straightforward implementation of this uses three copies of the LowMC circuit, we present a further optimization. The nature of the secret-sharing and circuit decomposition used in ZKB++ ensures that for each wire w in the circuit, the equality $w = w_1 \oplus w_2 \oplus w_3$, holds, where w_i is the share of party i. If we evaluate the circuit once in plain and store all intermediate values w, we can use only two instances of LowMC for signing and compute the shares of the third party $w_3 = w \oplus w_1 \oplus w_2$ whenever needed. This optimization allows us to implement the LowMC-MPC module using resources equivalent to only two LowMC circuit evaluations, while still being able to evaluate all players simultaneously. Additionally, we can precompute the plain evaluation of the LowMC circuit in parallel to the Seeds calculation at the beginning of the Picnic signing process and, therefore, do not slow down signing while using this optimization.

During signature verification, only two players perform the LowMC-MPC circuit evaluation; therefore, we naturally only require resources of about two LowMC circuit evaluations to perform all players in parallel.

SHAKE. In Picnic, instances of SHAKE are used for different purposes, both as a hash function with fixed output or as an extensible output function to generate pseudorandom tapes of arbitrary size. Therefore, we implemented a custom, flexible Keccak design, supporting many different configurations while maintaining efficiency and small hardware utilization.

Seeds, Tapes, and Commitments. In the beginning, one master seed is pseudorandomly generated and expanded into seeds for each of the T runs. We use three instances of SHAKE to expand the seeds for each player's current run into its random tape and three more instances to commit to the transcript of the current run for each player. We are capable of calculating the randomness required for run $t+1$ of Picnic's circuit decomposition in parallel to calculating the commitments of run t, reducing the overall number of clock cycles for signing and verification. However, due to limited routing freedom due to high resource utilization of our synthesized Picnic-L5 design, this optimization would significantly increase the critical path of the design and, therefore, this optimization is only used for Picnic-L1.

Challenge Generation (H3). Based on the Fiat-Shamir transformation, we instantiate the random oracle for the challenge generation using SHAKE. All commitments for all T runs are hashed together with some additional parameters to produce the challenge vector. Since the challenge vector consists of entries in $\{0, 1, 2\}$ to denote the player that is not revealed for this run, the H3 module takes care to filter the output bits of the SHAKE call according to the PICNIC specification.

Serialization and Deserialization. We also implemented small submodules to assemble the final signature as a byte array conforming to the PICNIC specification. For verification, we parse incoming signatures and store all the intermediate values of the opened views in the block RAM cells of the FPGA. These modules are implemented to be able to handle the variable signature length of PICNIC internally.

High-Level Design. We developed several different VHDL designs for PICNIC-L1-FS and PICNIC-L5-FS. We implemented a standalone version for message signing or signature verification only, as well as a version which is capable of doing both.

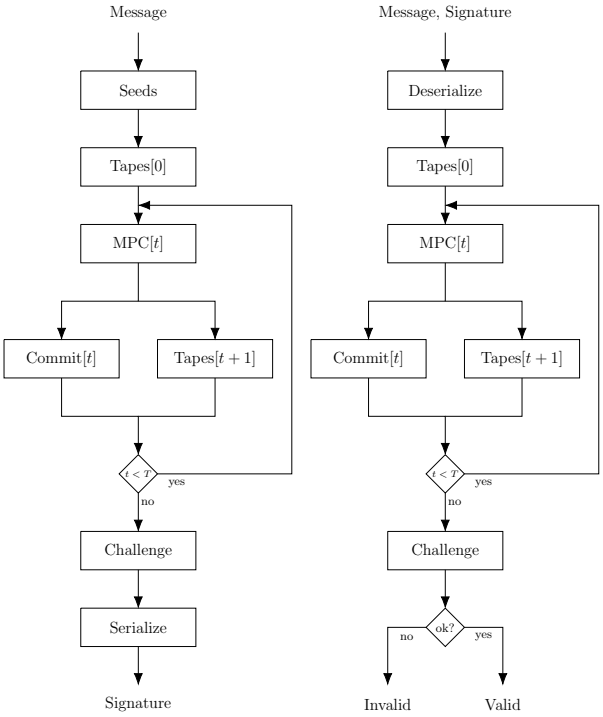


Fig. 3: High-level design of PICNIC signing (left) and verification (right).

The overall design of the implementations is a nested state machine, where the high-level design connects the inputs and outputs of all the described submodules. Figure 3 shows a diagram of the high-level design, with the signing process shown on the left side and the verification process on the right. In the designs which are capable of doing both signing and verification, both processes are implemented. Most of the submodules can be reused for both signing and verification, only the MPC module has to implement two different Sbox calculations, and the combined design has to include both the *Serialize* and *Deserialize* submodules. Therefore, the difference in hardware utilization between a sign-only design and a sign/verify design is quite low.

4 Evaluation

In the following, we evaluate and discuss the performance and hardware utilization of our design. We not only give cycle counts for signing and verification but also show the real-world performance of our designs by additionally synthesizing a PCIe wrapper around our Picnic cores and using them from a C library.

4.1 Hardware Utilization

First, we give an overview of the required hardware utilization of the different Picnic submodules, and then show the utilization of the developed coprocessors. The used FPGA, a Xilinx Kintex-7 board, has 203800 lookup-tables (LUTs), 407600 flip-flops (FF) and 445 BRAMs available.

Picnic *Submodules.* To give an overview of the costs of the individual submodules, we present their hardware utilization for Picnic-L1-FS and Picnic-L5-FS in Table 3. This table shows that the LowMC-MPC modules require by far the most hardware utilization. However, observe that the combined submodule which is able to do the LowMC-MPC encryption for both signing and verification only requires less than one percent more LUTs than the submodule which can only be used for signing. This is because we can reuse large parts of the circuit for both signing and verification, the only difference is in the Sbox layer, where the AND gates are evaluated.

Picnic *Coprocessors.* Table 4 compares the hardware utilization of the different submodules of the final coprocessors, including our 6 different Picnic cores synthesized for the Kintex-7. Our Picnic cores require a lot of LUTs on the used FPGA, especially the Picnic-L5-FS implementations. The PCIe/DMA Subsystem which connects the Picnic cores to the PCIe port of the used FPGA board adds about 22 000 additional LUTs to the design.

On the Artix-7, the picture is quite different, as we only have 133 800 LUTs, 267 600 flip-flops and 365 BRAMs available. Consequently, neither Picnic-L5 Picnic-L5-sign, nor Picnic-L5-verify fit on this board. The hardware utilization of the Artix implementations of L1 are depicted in Table 5.

Table 3: Hardware utilization for different parts of the L1 and L5 designs on Kintex-7.

	L1				L5			
Design Part	LUTs	%	FF	%	LUTs	%	FF	%
Keccak	3726	1.83 %	1606	0.39 %	3726	1.83 %	1606	0.39 %
Tapes (3× Keccak)	9574	4.70 %	5589	1.37 %	9420	4.62 %	9621	2.36 %
Commits (3× Keccak)	12221	6.00 %	5589	1.37 %	14160	6.95 %	6357	1.56 %
Seeds (1× Keccak)	5867	2.88 %	1846	0.45 %	8974	4.40 %	2640	0.65 %
H3 (1× Keccak)	7236	3.55 %	3641	0.89 %	8815	4.33 %	4085	1.00 %
Serialize	1962	0.96 %	125	0.03 %	1608	0.79 %	172	0.79 %
Deserialize	2025	0.99 %	125	0.03 %	2317	1.14 %	155	0.04 %
LowMC-MPC Sign	31837	15.62 %	3060	0.75 %	97066	47.63 %	5940	1.46 %
LowMC-MPC Verify	29756	14.60 %	1126	0.28 %	93959	46.10 %	2246	0.55 %
LowMC-MPC	32224	15.81 %	3061	0.75 %	98319	48.24 %	5958	1.46 %

Table 4: Hardware utilization for different parts of the coprocessor for Kintex-7.

Design Part	LUTs	%	FF	%	BRAM	%
PCIe/DMA	22216	10.90 %	22692	5.57 %	42.5	9.55 %
Picnic-L1	90037	44.18 %	23105	5.67 %	52.5	11.80 %
Picnic-L1-sign	76472	37.52 %	21061	5.17 %	52.5	11.80 %
Picnic-L1-verify	68614	33.67 %	16821	4.13 %	33.5	7.53 %
Picnic-L5	167530	82.20 %	33164	8.14 %	98.5	22.13 %
Picnic-L5-sign	149456	73.33 %	30441	7.47 %	98.5	22.13 %
Picnic-L5-verify	138547	67.98 %	24278	5.96 %	62.5	14.04 %

Table 5: Hardware utilization on Artix-7.

Design Part	LUTs	%	FF	%	BRAM	%
Picnic-L1	90037	67.29 %	23105	8.63 %	52.5	14.38 %
Picnic-L1-sign	76472	57.15 %	21061	7.87 %	52.5	14.38 %
Picnic-L1-verify	68614	51.28 %	16821	6.29 %	33.5	9.18 %

Critical Path. The critical path of the synthesized design is across the matrix multiplications in LowMC's linear layer and round key schedule, due to the high number of constants involved. But we also observed, that since the Picnic-L5 design has a considerable hardware utilization, the synthesizer has much

Table 6: Clock Cycles per Submodule.

Design Part	Picnic-L1-FS	Picnic-L5-FS
LowMC-MPC	40	76
Tapes	51	75
Commits	51	100
Seeds	1 732	7 904
H3 (absorb)	6 490	26 220
Deserialize	1 per 128 bit	1 per 128 bit
Serialize	1 per 128 bit	1 per 128 bit
$T\times$ LowMC-MPC	8 760	31 844
$T\times$ Tapes	11 169	31 425
$T\times$ Commits	11 169	41 900

less freedom in routing the design and, therefore, naturally produces long paths between registers. These long paths make it very difficult to optimize the design for high frequencies.

4.2 Clock Cycles

Table 6 lists the number of clock cycles each submodule of our Picnic implementation requires. The LowMC-MPC module performs the evaluation of a round in two clock cycles and, therefore, requires $2 \cdot r$ cycles in total. Evaluating a round in one cycle would have drastically increased the critical path of the design since two matrix multiplications (linear layer and round key schedule) would have been performed sequentially in this case.

Our Keccak implementation performs one round of the state transformation function during one clock cycle, which leads to 24 cycles for one absorbing/squeezing phase. The number of absorbing/squeezing phases, therefore, determines the number of clock cycles required for the *Tapes*, *Commits*, *Seeds*, and the first part of the *H3* submodules. The duration of the second part of the *H3* submodule depends on the generated challenge and differs for every signature.

In Picnic we have T runs of the FS transformed ZKB++ proof system, in contrary to the *Seeds*, *H3*, *Serialize* and *Deserialize* modules which are only required once. In Table 6 we, therefore, also show the overall runtime of each submodule involved in the proof creation and it can be seen, that the proof system dominates the overall runtime of the signature creation and verification process.

4.3 Benchmarks

To verify the performance characteristics of our implementation, we compared the runtime of the coprocessors running on a Kintex-7 board to the state-of-the-

Table 7: Runtime comparison of the coprocessors on benchmark platform A.

Coprocessor	clock frequency (MHz)	clock cycles	FPGA runtime (ms)	C-Access runtime (ms)
PICNIC-L1-SIGN	125	≈ 31300	0.250	0.349
PICNIC-L1-VERIFY	125	≈ 29600	0.237	0.395
PICNIC-L5-SIGN	125	≈ 154500	1.236	1.383
PICNIC-L5-VERIFY	125	≈ 146600	1.173	2.128

art optimized software implementations of PICNIC. The platforms used for the benchmarks are as follows:

Platform A Intel i7-960, 3.2 GHz with 16 GB RAM, Debian 9
Platform B Intel i7-4790, 3.6 GHz with 16 GB RAM, Ubuntu 18.04.1
Platform C Intel E31230, 3.2 GHz with 8 GB RAM, Ubuntu 18.04.2

We used platform A to test our coprocessors, platforms B and C were used in the PICNIC design document [19] to test their optimized software implementations.

Table 7 shows the average runtime of the developed coprocessors for signing and verification. The column *FPGA runtime* is the calculated time resulting from the clock frequency and the number of clock cycles (including 1 cycle per 128 bit of data transmission) and therefore is the actual runtime of the FPGA. The column *C-Access runtime* is the measured runtime using our developed C library on platform A.

As Table 7 shows, the C library developed to interface with the coprocessor adds some overhead to the signing and verification process. For signing, the overhead is about 0.1 ms in runtime, but for verification, the overhead is a bit larger. Especially for PICNIC-L5-FS the measured runtime is much bigger than the raw verification runtime of the coprocessor. We suspect that this is due to the driver for the PCIe/DMA Subsystem being slower for writing large amounts of data, like the PICNIC-L5-FS signature, from the PC to the FPGA board and that this overhead could be optimized further.

For comparison, Table 8 shows the runtime of the optimized implementation of PICNIC in C and an optimized version which uses processor-specific compiler intrinsics on two different benchmark platforms as described in the official PICNIC design document [19]. This table shows, that the runtime of PICNIC highly depends on the underlying hardware and if the CPU supports *single instruction, multiple data* (SIMD) instruction sets, like SSE2 and AVX2, which further improve execution time. However, in any case, our developed coprocessors are faster than the corresponding software counterparts and do not rely on specific CPU instructions. For PICNIC-L1-FS signing is ≈ 4 times faster than the fastest software implementation, verification is ≈ 3 times faster. For PICNIC-L5-FS our implementations are ≈ 4 times faster for signing and ≈ 2.3 times faster for verification. For CPUs which do not support AVX2 instructions and for portable C-only implementations the speedup of our coprocessors is even more significant.

Table 8: Runtime comparison of optimized software implementations [19].

Platform	Parameters	using SIMD	Sign	Verify
B	Picnic-L1-FS	✓	1.44 ms	1.15 ms
B	Picnic-L5-FS	✓	5.87 ms	4.92 ms
B	Picnic-L1-FS	✗	2.82 ms	2.34 ms
B	Picnic-L5-FS	✗	12.37 ms	10.59 ms
C	Picnic-L1-FS	✓	4.20 ms	3.40 ms
C	Picnic-L5-FS	✓	17.67 ms	14.67 ms
C	Picnic-L1-FS	✗	4.41 ms	3.56 ms
C	Picnic-L5-FS	✗	19.52 ms	16.81 ms

Table 9: Comparison of FPGA implementations (modified from [7]).

	Security			Area	f	t
Scheme	Classic	PQ	FPGA	LUT/FF/BRAM	MHz	ms
SPHINCS-256 [7]	256	128	K7	19067/38132/36	525	1.53
SPHINCS+-128 [9]	128	64	V7	11438/3335/?	100	9.38
BLISS-IV [43]	192	?	S6	6438/6198/7	135	0.35
ECDSA-256 [6]	128	✗	V7	6816/4442/0	225	1.49
ECDSA-256 [33]	128	✗	V4	34869/32430/176	375	0.04
ECDSA-521 [6]	256	✗	V7	8273/7689/0	161	5.02
RSA-2048 [47]	112	✗	V7	3558 slices/0	399	5.68
Picnic-L1-FS	128	64	K7	90037/23105/52.5	125	0.25
Picnic-L5-FS	256	128	K7	167530/33164/98.5	125	1.24

4.4 Comparison to FPGA Implementations of Other Signature Schemes

To put our FPGA implementation in context of other signature schemes, we compare our Picnic coprocessors to implementations of ECDSA [6] and RSA coprocessors [47] as well as implementations of SPHINCS-256 [7] and BLISS-IV [43]. Table 9 compares several different FPGA implementations of various signature schemes, the runtime for signing t is calculated from the clock frequency and the number of clock cycles and therefore does not take the overhead of any transmission of data via a C-program into account. Thus this value compares to the column *FPGA runtime* of Table 7.

As Table 9 shows, our Picnic-L5-FS coprocessors, which have the same security level as a SPHINCS-256 [7] coprocessor, have a slightly better runtime for signing on the Kintex-7 (K7) FPGA. Similar, for the SPHINCS+ design obtained from the high-level synthesis design flow [9], our coprocessor has a significant better runtime at the cost of higher hardware utilization. The im-

plementations of the traditional signature schemes RSA [47] and ECDSA [6] on a Virtex-7 (V7) FPGA are also slower than our coprocessors. The ECDSA implementation in [33] occupies more area but uses high parallelism to drastically increase their throughput. The implementation of BLISS-IV [43], another post-quantum signature scheme based on lattices, on a Spartan-6 (S6) FPGA is very efficient regarding area and runtime for signing. However, it has a lower security level, and its security against a quantum adversary is not as well understood as for schemes based on symmetric primitives like SPHINCS and Picnic.

However, even though our coprocessors are very competitive with regards to signing times, the hardware utilization is significantly higher in comparison to implementations of other signature schemes. This is due to the nature of Picnic relying on a high number of different Keccak and LowMC primitives, where especially the LowMC instances have a high hardware utilization on their own. In comparison, the coprocessor of SPHINCS-256, a hash-based post-quantum signature scheme, can be built efficiently using only one pipelined ChaCha12 instance and one instance of BLAKE-256 [8] and as a result, requires less hardware utilization [7].

4.5 Evaluation of the LowMC Pipeline Design

Finally, we evaluate our pipelined design. After r cycles, the design is capable of producing one ciphertext per cycle (cf. Section 3.2), a feature which is of particular interest for high throughput use cases. We compare our coprocessor ($f = 125\,\text{MHz}$) for a LowMC instance with 128 bit block size and full data complexity to AES-128 accelerated with the AES-NI instruction set [35]. For this comparison we choose a LowMC instance with $n = 128$, $k = 128$, $m = 25$, $r = 11$. This instance provides a trade-off between costs in the linear layer and the number of AND gates. The comparison of the coprocessor, including C-access times, the raw FPGA runtime, the SIMD-optimized LowMC software implementation and AES-NI is depicted in Table 10. These benchmarks were recorded on a PC running Ubuntu 16.05 with an Intel i7-4790 CPU, 3.6 GHz. As the table shows, the coprocessor speeds up encryption by a factor of ≈ 84 compared to the LowMC software implementation, and when considering the C-access time, the improvement is still up to a factor of ≈ 14. Compared to AES-NI, the raw performance on the FPGA is better by a factor of ≈ 2.75, but the access time adds significant overhead. Therefore, we expect this design to render LowMC an alternative for PSI protocols [37] or database joins on secret shared data [40].

This speed up direct translates to the same performance gain in the setup phase of the PSI protocol as proposed in [37]. Thus, the excellent performance of our coprocessor makes it feasible to use LowMC in the PSI protocol. Thereby, the PSI protocol can profit from reduced communication overhead during the online phase due to the reduced multiplicative complexity of LowMC without requiring the tradeoff of having a slower setup phase.

Table 10: Performance of LowMC($n = 128, k = 128, m = 25, r = 11$) implemented in software and in our pipeline coprocessor, as well as AES-NI.

		LowMC			AES
# Encryptions	Size	FPGA-Raw	FPGA-C	Software	AES-NI
2^{20}	16 MB	0.008s	0.046s	0.677s	0.022s
2^{24}	256 MB	0.134s	0.771s	10.78s	0.359s
2^{26}	1024 MB	0.537s	3.11s	43.31s	1.436s
2^{28}	4096 MB	2.15s	12.57s	182.65s	5.743s

5 Reducing the Hardware Utilizations

The large size of the constants needed for LowMC is one of the limiting factors to implement Picnic on FPGAs. Even after applying the optimizations to the round key and linear layer computations, the constants are still too large to fit an implementation on an Artix-7 board. To fit an implementation of Picnic suitable for the 128-bit post-quantum security level on this board, different LowMC instances with fewer rounds could be selected. Conversely, as this change requires the number of Sboxes to be increased to retain the security guarantees, the signature size will increase. In addition to fitting Picnic on smaller FPGA boards, the performance of the optimized implementations would also improve, since fewer rounds are required to achive the same level of security when more Sboxes are used. Alternatively, further improvements are required to reduce the size of LowMC constants. We envision multiple alternatives that could make this possible.

The current design of Picnic was chosen to have an acceptable trade-off between area and runtime and, therefore, evaluates the LowMC-MPC simulation concurrently for all three players by using two instances of the LowMC matrices. By doing the MPC simulation consecutively, we would be able to reduce the instances used to only one and reduce the hardware utilization. However, this optimization would result in more clock cycles per LowMC-MPC rounds and longer critical paths after synthesis reducing the clock frequency, and, therefore, would produce a very high performance penalty of at least a factor 2 if not more.

Another possibility to reduce the hardware utilization by modifying our design would be to reuse Keccak instances for different purposes in the design. However, this would again result in longer critical paths after synthesis, and since our Keccak design is very small in comparison to the LowMC design, the resulting performance penalty is too big in comparison to the actual reduction of the hardware utilization.

The use of LowMC in Picnic is relatively unique in the sense that it uses LowMC instances with low data complexity. Only recently, LowMC in this setting has seen more security analysis [44], leading to LowMC version 3 with a higher number of rounds. While the higher number of rounds on its own is not a problem for the FPGA implementation, the size of the constants also increases as

Table 11: Hardware utilization (LUTs) with reduced LowMC.

Design Part	LUTs	Improvement	Utilization	
			Kintex-7	Artix-7
LowMC MPC-L1	17751	44.91 %	8.71 %	13.27 %
LowMC MPC-L5	47615	51.57 %	23.36 %	35.59 %
Picnic-L1	75662	15.97 %	37.13 %	56.55 %
Picnic-L1-sign	62272	18.57 %	30.56 %	46.54 %
Picnic-L1-verify	55321	19.37 %	27.14 %	41.35 %
Picnic-L5	121299	27.60 %	59.52 %	90.66 %
Picnic-L5-sign	103688	30.62 %	50.88 %	77.49 %
Picnic-L5-verify	92910	32.94 %	45.59 %	69.44 %

more and more unique matrices are required. However, new designs [30] that are also optimized for a low multiplicative complexity make use of a single matrix for the linear layer. We propose to apply this idea also to LowMC, that is, the same uniformly random matrix is re-used for all linear layers.[6] Thereby we can significantly reduce the hardware utilization as can be seen in Table 11. With this change, the Picnic-L5 design fits on the Artix-7.

Furthermore, with this change in place, one could go a step further and remove the constants from the implementation altogether. The matrices and round constants could then be derived from the LFSR as specified in the LowMC instance generation algorithm. It would be necessary to store the intermediate states of the LFSR where it is known to produce invertible matrices, though, but then no invertible checks would need to be implemented. While deriving the matrices during runtime would come with a performance penalty, we expect it the reduce the hardware utilization significantly.

Alternatively, LowMC could be replaced by recently proposed cipher designs such as GMiMC [2]. Similarly to LowMC, GMiMC – and in particular its ERF variant – can be parameterized for the low data complexity scenario. It can be parameterized in a way leading to roughly similar sized signatures with better performance (in software). However, for an FPGA implementation, we expect it to use a lot less area since the size of the constants is significantly smaller. For the GMiMC instance over $\mathrm{GF}(2^{17})$ with 63 rounds, the constants would consist of only 63 field elements in total. The additional multipliers required for $\mathrm{GF}(2^{17})$ are cheap [25] when compared to the size of LowMC matrices.

[6] The LowMC designers confirmed in private communication that they do not expect this change to enable a new attack vector. However, more security analysis on this case would be required before this can be integrated into Picnic itself.

6 Conclusion

In this work, we presented two LowMC designs for FPGAs. The first design relying on a simple state machine shows the feasibility of implementing the post-quantum signature scheme Picnic on FPGA platforms. The resulting FPGA design can sign messages for the L1 security level in ≈ 31300 cycles and verify signatures in ≈ 29600 cycles. Using our concrete FPGA board, a Xilinx Kintex-7 FPGA KC705 evaluation kit, this allows signing of a message using a C library communicating with our board connected via PCIe in 0.35 ms.

Acknowledgements. This work was partially supported by the EU's Horizon 2020 ECSEL Joint Undertaking project SECREDAS under grant agreement n°783119, by the European Research Council (ERC) under Horizon 2020 grant agreement n°681402, by EU's Horizon 2020 project Safe-DEED under grant agreement n°825225, and by the IoT4CPS project which is partially funded by the "ICT of the Future" Program of the FFG and the BMVIT. D. Kales was supported by iov42 Ltd.

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status report on the first round of the nist post-quantum cryptography standardization process (2019). https://doi.org/10.6028/NIST.IR.8240
2. Albrecht, M.R., Grassi, L., Perrin, L., Ramacher, S., Rechberger, C., Rotaru, D., Roy, A., Schofnegger, M.: Feistel structures for mpc, and more. In: ESORICS (2). LNCS, vol. 11736, pp. 151–171. Springer (2019)
3. Albrecht, M.R., Grassi, L., Rechberger, C., Roy, A., Tiessen, T.: Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In: ASIACRYPT (1). LNCS, vol. 10031, pp. 191–219 (2016)
4. Albrecht, M.R., Hanser, C., Höller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Implementing rlwe-based schemes using an RSA co-processor. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(1), 169–208 (2019)
5. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: EUROCRYPT (1). LNCS, vol. 9056, pp. 430–454. Springer (2015)
6. Amiet, D., Curiger, A., Zbinden, P.: Flexible fpga-based architectures for curve point multiplication over gf(p). In: DSD. pp. 107–114. IEEE Computer Society (2016)
7. Amiet, D., Curiger, A., Zbinden, P.: Fpga-based accelerator for post-quantum signature scheme SPHINCS-256. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 18–39 (2018)
8. Aumasson, J., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. In: ACNS. LNCS, vol. 7954, pp. 119–135. Springer (2013)
9. Basu, K., Soni, D., Nabeel, M., Karri, R.: NIST post-quantum cryptography- A hardware evaluation study. ePrint **2019**, 47 (2019)
10. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. ePrint **2013**, 404 (2013)

11. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: EUROCRYPT (1). LNCS, vol. 9056, pp. 368–397. Springer (2015)
12. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs$^+$ signature framework. In: CCS. pp. 2129–2146. ACM (2019)
13. Boneh, D., Dagdelen, Ö., Fischlin, M., Lehmann, A., Schaffner, C., Zhandry, M.: Random oracles in a quantum world. In: ASIACRYPT. LNCS, vol. 7073, pp. 41–69. Springer (2011)
14. Boneh, D., Eskandarian, S., Fisch, B.: Post-quantum EPID signatures from symmetric primitives. In: CT-RSA. LNCS, vol. 11405, pp. 251–271. Springer (2019)
15. Bouillaguet, C., Derbez, P., Fouque, P.: Automatic search of attacks on round-reduced AES and applications. In: CRYPTO. LNCS, vol. 6841, pp. 169–187. Springer (2011)
16. Boyar, J., Matthews, P., Peralta, R.: Logic minimization techniques with applications to cryptology. J. Cryptology **26**(2), 280–312 (2013)
17. Canteaut, A., Carpov, S., Fontaine, C., Lepoint, T., Naya-Plasencia, M., Paillier, P., Sirdey, R.: Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In: FSE. LNCS, vol. 9783, pp. 313–333. Springer (2016)
18. Chailloux, A.: Quantum security of the fiat-shamir transform of commit and open protocols. ePrint **2019**, 699 (2019)
19. Chase, M., Derler, D., Goldfeder, S., Katz, J., Kolesnikov, V., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Wang, X., Zaverucha, G.: The picnic signature scheme design document (version 2) (2019), `https://github.com/microsoft/Picnic/blob/master/spec/design-v2.0.pdf`
20. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: ACM CCS. pp. 1825–1842. ACM (2017)
21. Derler, D., Ramacher, S., Slamanig, D.: Generic double-authentication preventing signatures and a post-quantum instantiation. In: ProvSec. LNCS, vol. 11192, pp. 258–276. Springer (2018)
22. Derler, D., Ramacher, S., Slamanig, D.: Post-quantum zero-knowledge proofs for accumulators with applications to ring signatures from symmetric-key primitives. In: PQCrypto. LNCS, vol. 10786, pp. 419–440. Springer (2018)
23. Dinur, I., Kales, D., Promitzer, A., Ramacher, S., Rechberger, C.: Linear equivalence of block ciphers with partial non-linear layers: Application to lowmc. In: EUROCRYPT (1). LNCS, vol. 11476, pp. 343–372. Springer (2019)
24. Don, J., Fehr, S., Majenz, C., Schaffner, C.: Security of the fiat-shamir transformation in the quantum random-oracle model. In: CRYPTO (2). LNCS, vol. 11693, pp. 356–383. Springer (2019)
25. El-Razouk, H., Reyhani-Masoleh, A.: New bit-level serial GF (2^{m}) multiplication using polynomial basis. In: ARITH. pp. 129–136. IEEE (2015)
26. Ferozpuri, A., Farahmand, F., Dang, V., Sharif, M.U., Kaps, J.P., Gaj, K.: Hardware API for Post-Quantum Public Key Cryptosystems (2018), `https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf`
27. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO. LNCS, vol. 263, pp. 186–194. Springer (1986)
28. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions. In: TCC. LNCS, vol. 3378, pp. 303–324. Springer (2005)

29. Giacomelli, I., Madsen, J., Orlandi, C.: Zkboo: Faster zero-knowledge for boolean circuits. In: USENIX Security Symposium. pp. 1069–1083. USENIX Association (2016)
30. Grassi, L., Kales, D., Khovratovich, D., Roy, A., Rechberger, C., Schofnegger, M.: Starkad and poseidon: New hash functions for zero knowledge proof systems. ePrint **2019**, 458 (2019)
31. Grassi, L., Rechberger, C., Rotaru, D., Scholl, P., Smart, N.P.: Mpc-friendly symmetric key primitives. In: ACM CCS. pp. 430–443. ACM (2016)
32. Grosso, V., Leurent, G., Standaert, F., Varici, K.: Ls-designs: Bitslice encryption for efficient masked software implementations. In: FSE. LNCS, vol. 8540, pp. 18–37. Springer (2014)
33. Güneysu, T.: Utilizing hard cores of modern FPGA devices for high-performance cryptography. J. Cryptographic Engineering **1**(1), 37–55 (2011)
34. Howe, J., Oder, T., Krausz, M., Güneysu, T.: Standard lattice-based key encapsulation on embedded devices. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 372–393 (2018)
35. Intel Corporation: Securing the enterprise with intel® aes-ni (2010), `https://www.intel.com/content/dam/doc/white-paper/enterprise-security-aes-ni-white-paper.pdf`
36. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: STOC. pp. 21–30. ACM (2007)
37. Kales, D., Rechberger, C., Schneider, T., Senker, M., Weinert, C.: Mobile private contact discovery at scale. In: USENIX Security Symposium. pp. 1447–1464. USENIX Association (2019)
38. Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: ACM CCS. pp. 525–537. ACM (2018)
39. LowMC: Official LowMC Github Repository, `https://github.com/LowMC/lowmc`
40. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins for secret shared data. ePrint **2019**, 518 (2019)
41. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In: EUROCRYPT. LNCS, vol. 6632, pp. 69–88. Springer (2011)
42. Naehrig, M., Lauter, K.E., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: CCSW. pp. 113–124. ACM (2011)
43. Pöppelmann, T., Ducas, L., Güneysu, T.: Enhanced lattice-based signatures on reconfigurable hardware. In: CHES. LNCS, vol. 8731, pp. 353–370. Springer (2014)
44. Rechberger, C., Soleimany, H., Tiessen, T.: Cryptanalysis of low-data instances of full lowmcv2. IACR Trans. Symmetric Cryptol. **2018**(3), 163–181 (2018)
45. Rotaru, D., Smart, N.P., Stam, M.: Modes of operation suitable for computing on encrypted data. IACR Trans. Symmetric Cryptol. **2017**(3), 294–324 (2017)
46. Roy, D.B., Mukhopadhyay, D.: Post quantum ECC on FPGA platform. ePrint **2019**, 568 (2019)
47. San, I., At, N.: Improving the computational efficiency of modular operations for embedded systems. Journal of Systems Architecture - Embedded Systems Design **60**(5), 440–451 (2014)
48. Sasdrich, P., Güneysu, T.: Implementing curve25519 for side-channel-protected elliptic curve cryptography. TRETS **9**(1), 3:1–3:15 (2015)
49. Unruh, D.: Non-interactive zero-knowledge proofs in the quantum random oracle model. In: EUROCRYPT (2). LNCS, vol. 9057, pp. 755–784. Springer (2015)
50. Wang, W., Szefer, J., Niederhagen, R.: Fpga-based niederreiter cryptosystem using binary goppa codes. In: PQCrypto. LNCS, vol. 10786, pp. 77–98. Springer (2018)

51. Werner, M., Unterluggauer, T.: Transparent memory encryption and authentication, https://github.com/IAIK/memsec
52. Werner, M., Unterluggauer, T., Schilling, R., Schaffenrath, D., Mangard, S.: Transparent memory encryption and authentication. In: FPL. pp. 1–6. IEEE (2017)