Table 10: Area and time results (written as X/Y for GF256/GF251) for sign_online module for all security levels and underlying arithmetic fields targeting the Xilinx Artix 7 `xc7a200t` FPGA.

Param Sets	FPGA Utilisation					Freq.	Latency	Time	TAP
	Slices	LUT	FF	DSP	BRAM	(*MHz*)	(*Mcycles*)	(*ms*)	
	($\times 10^3$)	($\times 10^3$)	($\times 10^3$)	($\times 10$)					($\times 10^3$)
L1	0.8/1.6	2.0/4.3	1.8/4.9	0/6	4/4	265/172	6.7/21.4	25/124	21/207
L3	2.3/2.5	4.1/6.6	3.6/7.9	0/12	6/6	261/171	7.7/24.5	30/143	70/366
L5	2.4/3.1	4.2/8.0	4.0/9.9	0/15	12.5/12.5	262/174	17.4/45.2	67/260	163/807

6.3 Interleaved sign_offline and sign_online

As noted in Section 6, Algorithm 6a, and Algorithm 6b, we split the SDitH signature generation algorithm into two phases, *offline* and *online*. We do so to hide/mask the cycles taken by the offline part of the signature generation. Figure 8 shows the hardware block design of our signature_generation module with the interleaving capability. As shown in the timing diagram in Figure 8, our module can handle the signing of two messages in an interleaved fashion while using a single SHAKE module. The process of interleaved signature generation is as follows: the first message enters the offline part, and after the offline processing, if the online part is part is available, all the data is buffered into the mem_buffer shown in Figure 8. Once the data becomes available, the sign_online part is started. While the sign_online gets started, the sign_offline loads a new message and starts processing the new message, but the new data is not added to the mem_buffer until again the sign_online becomes available.

In addition to this, keeping in mind our area-optimisation target for our hardware design, we only use one SHAKE module to fulfill the hashing and pseudo-random generation requirements for both sign_online and sign_offline parts. The sharing is handled by the shake_scheduler logic. Our shake_scheduler is able to accomplish the sharing of the SHAKE module without any additional penalty in terms of clock cycles. This is possible because of the way we split the SDitH signature generation algorithm. We note that due to our splitting, the amount of clock cycles required for the PartyComputation operation (in sign_online) is higher than that of the whole sign_offline part, this way every time the sign_online part requires the SHAKE module it is available.

We limit our interleaved signing to two messages mainly because of the high memory requirements posed by the SDitH signature generation algorithm. We note from Algorithm 6a, in the whole signature generation process only public matrix H' generation operation for GF251 in `sign_offline` is of variable time. However, since the `sign_offline` and `sign_online` modules work is parallel and `sign_online` takes more clock cycles compared to `sign_offline` (as shown in Table 9 and Table 10) this variable time behavior is completely masked and the overall `sign_interleaved` module still remains constant-time. From Table 11, we note that interleaving operation adds approximately 30-60% of additional BRAM based on the choice of the security level. We also note that this interleaving is an option, thus, if our sign_online and sign_offline modules are glued together without the memory_buffer, it can still work as the regular signature generation hardware module without the interleaving capability.

7 Signature Verification

The signature verification module takes the public key input (seed_H, y), signature, σ, which consists of the salt, hash (h_2), τ sibling path views, τ plain broadcast values (broad_plain), and commits of revealed views (com), and a message m as inputs and generates a valid signal as the output if the signature has been verified. Algorithm 7 shows the signature verification algorithm and Figure 4 shows the respective block diagram of our hardware module.

Our hardware design first starts with expanding the H' matrix using the SFTM SD instance module described in Section 3.4. As specified in Section 3.4, this operation is constant-time when the underlying arithmetic is GF256 and is variable time for GF251 due to rejection sampling. But, this is acceptable because H' is a public matrix and this way of implementation is compliant with the reference implementation. Apart from this operation, other underlying operations in the signature verification module are constant-time. The SD module waits until input_mshare is computed, performs the matrix-vector multiplication. After that, we expand the Fiat-Shamir hash h_2 into a view-opening challenge (i) using the ExpandViewChallenge module described in Section 3.2.4 and store this in a BRAM. Then, each i

Table 11: Area and time results (written as X/Y for GF256/GF251) for sign_interleaved module for all security levels and underlying arithmetic fields targeting the Xilinx Artix 7 xc7a200t FPGA.

Param	**FPGA Utilisation**					**Freq.**	**Latency**	**Time**	**TAP**
Sets	**Slices**	**LUT**	**FF**	**DSP**	**BRAM**	(MHz)	$(Mcycles)$	(ms)	
	$(\times 10^3)$	$(\times 10^3)$	$(\times 10^3)$	$(\times 10)$					$(\times 10^3)$
L1	2.0/3.2	5.0/8.4	4.2/9.1	0/12	99/99	240/172	6.7/21.4	28/124	56/403
L3	4.1/5.0	8.6/13.1	7.1/15.3	0/24	246/246	244/171	7.7/24.5	31/143	130/727
L5	4.3/6.1	9.1/15.9	7.8/18.9	0/30	353/353	241/174	17.4/45.2	72/260	314/1,599

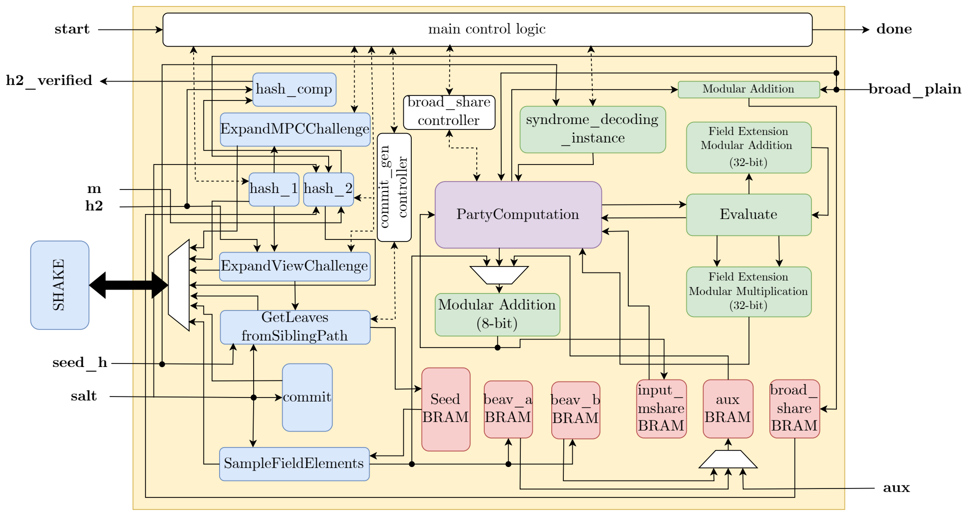

Figure 4: Hardware block design for Signature Verification Module interfaced with SHAKE module.

value is chosen from BRAM along with the salt and view (consisting of *path* and aux) and fed into the GetLeavesFromSiblingPath module described in Section 3.2.6 and all missing leaf seeds from the sibling path except the indexed one are generated and stored in a Seed BRAM. Each seed from the Seed BRAM, along with salt, 16-bit execution index, and 16-bit share index are fed into the Commit module to generate all the missing commits. These commit values are store in a BRAM inside the Commit module. The hash_1 module then uses the seed_H, public-key y, salt, and commits to generate the Fiat-Shamir hash h_1. h_1 is loaded into the ExpandMPCChallenge module to generate τ chal $= (r, \varepsilon)$ values.

Afterwards, we repeat this process τ times following operations where our SampleFieldElements module is fed with each seed from Seed BRAM and the input_share, beaver triples (*beav_a*, *beav_b* *beav_c*), and input_mshare values are then generated. Here, storing or accumulating all input_share values is not necessary because if we recall in the signature generation algorithm, Algorithm 6a, input_shares are mainly used for generating broad_plain, which is already fed as input to our verification module. The reason we generate input_shares is to compute the input_mshares which we generate using the input_mshare add & store pool shown in Figure 4 and to compute the beaver triples. After that, the PartyComputation module (described in Section 4.3) is fed with input_mshare, input broad_plain values, chal used to generate the broad_shares if input_mshare and the part of the public-key (y) to generate the broad_shares. We repeat the operation from these broad_shares which are stored in a broad_share BRAM.

Finally, we feed our Hash_2 module described in Section 3.2.8 with all the broad_share values, broad_plains values, salt, h_1, and the message m and h_2' hash is computed. The generated h_2' is compared against input h_2 using the hash_comp module shown in Figure 4 to generate the h2_verified output. h2_verified is high if $h_2' == h_2$ if not it stays low. From Table 12, we note that the memory utilisation is comparatively lower here mainly because the verification does not have the ability to be split into offline and online phases like in signature generation.

Table 12: Area and time results (written as X/Y for GF256/GF251) for sign_verification module for all security levels and underlying arithmetic fields targeting the Xilinx Artix 7 `xc7a200t` FPGA.

Param	**FPGA Utilisation**					**Freq.**	**Latency**	**Time**	**TAP**
Sets	**Slices**	**LUT**	**FF**	**DSP**	**BRAM**	(*MHz*)	(*Mcycles*)	(*ms*)	
	$(\times 10^3)$	$(\times 10^3)$	$(\times 10^3)$	$(\times 10)$					$(\times 10^3)$
L1	1.5/2.5	4.7/6.6	3.1/5.8	0/6	57/57	240/172	8.6/23.4	36/136	66/350
L3	2.5/3.2	6.5/8.5	4.9/8.9	0/12	95/95	244/171	10.9/19.3	44/112	113/372
L5	2.6/3.8	6.7/10.1	5.3/10.8	0/15	143/143	241/174	24.9/30.1	103/173	269/674

8 Comparisons to Related Works

In most other software and hardware designs of NIST PQC candidates, SHAKE is known to be a bottleneck. But in our area optimised hardware implementation of SDitH primitives we note that the bottleneck is not the SHAKE-256 but the polynomial evaluation module (Evaluate) which contributes to 99% clock cycles in sign (sign_online) and 70%-90% clock cycles in verification depending on the choice of security level and underlying arithmetic field. This adds a distinctive elements to SDitH and its hardware design. Additionally, its feature of being able to be split into offline and online phases illustrates its potential of being useful in many use cases, setting it apart from other NIST PQC candidates.

Comparisons to PQC signatures in Hardware In Table 14 we provide comparison of our design with the (to the best of our knowledge) state-of-the-art hardware implementations of Picnic [KRR+20], SPHINCS$^+$ [ALC+20], Dilithium [ZZW+22], and LESS [BWM+23] post-quantum signature schemes. From the tables, we note that only our SDitH implementation, Dilithium, and LESS implement all three primitives of the signature algorithm (key generation, sign, and verify). Whereas the SPHINCS$^+$ [ALC+20] implementation only presents signing and Picnic [KRR+20] implements only sign and verify.

From Table 14, we highlight that our SDitH-GF256 hardware implementation is of the smallest area footprint when compared to all other designs. Our SDitH-GF251 also uses less area but uses DSP resources for optimising the underlying arithmetic operations. However, our hardware designs use significant BRAM as it is unavoidable due to the nature of the SDitH signature scheme. When comparing the overall performance we note that Dilithium clearly outperforms all other designs. However, it may not be fair to compare the lattice-based schemes against those using MPCitH. A more relevant comparison would be with Picnic, in which case our design uses much less area while implementing all primitives. While we acknowledge that the time taken by the Picnic design to sign and verify is better compared to that of our design, the Picnic implementation uses a reduced data complexity design using a LowMC, compared to the more conservative code-based hardness assumption in SDitH.

From Table 13, the results of this work result in a hardware design with a drastic reducing in clock cycles compared to the optimised AVX2 software implementation, in the range of 2-4x for most operations. This is effectively due to how amenable SDitH is to hardware, its arithmetic types, its use of powers-of-two arithmetic, and its ability to parallelise many of its operations like the d-splitting and the offline/online stage split in signature generation.

Our key generation outperforms software drastically, ranging between 11-17× reduction in *runtime*, this is all while the software implementation has a 16× faster clock speed. We achieve this due to the design of SampleWitness, which allows us to optimally perform sampling and arithmetic operations in hardware, which is not as easily done in software. We also note that while our hardware design outperforms software by 2-3.4× in terms of signature generation cycles and 1.4-2.1× in terms of signature verification cycles, our design is slower when it comes to *runtime* comparison. The reason for this is threefold: (i) the operating frequency of the FPGA is much lower versus the processor running the software, (ii) as specified in Section 4.1, in the optimised software reference implementation of SDitH, all possible outcomes of the modular exponentiation are precomputed and stored in large lookup tables which was not possible in the hardware design due to the resource constraints, and (iii) for all the randomness generation and hashing requirements, the software implementation takes advantage of the optimised AVX2 instructions to run four Keccak (SHAKE) instructions in parallel, which would also not be feasible in hardware since our target was area-optimised design.

We also observe that key generation is faster for GF251 than for GF256, which is the opposite of the trend we observe in the case of our hardware implementation. This is due to the fact that, in software

Table 13: Performance comparison of our SDitH hardware designs with the optimised SDitH software implementation (written as X/Y for GF256/GF251).

Parameter	**KeyGen**		**Sign**		**Verify**	
Sets	**Latency**	**Time**	**Latency**	**Time**	**Latency**	**Time**
	(*Kcycles*)	(*ms*)	(*Mcycles*)	(*ms*)	(*Mcycles*)	(*ms*)
Our Hardware Design, Artix 7 (xc7a200t), Freq = 164 MHz						
L1	55.1/57.1	0.33/0.34	6.7/21.4	41.03/130.76	8.6/23.4	52.98/142.70
L2	46.0/47.0	0.28/0.29	7.7/24.5	47.16/149.98	10.9/19.3	66.84/117.76
L3	83.8/86.7	0.51/0.52	17.4/45.2	106.60/276.07	24.9/30.1	152.10/183.57
Reference Software [AFG+23], Intel Xeon E-2378, Freq = 2.6 GHz						
L1	–	4.12/2.70	13.4/22.1	5.18/8.51	12.5/21.2	4.81/8.16
L2	–	4.89/3.31	30.5/51.1	11.77/19.72	27.7/49.0	10.68/18.89
L3	–	8.75/5.93	59.2/94.8	22.86/36.56	54.4/91.3	20.98/35.33

implementation, they are able to use Galois-Field-New-Instructions[9] (GFNI) for GF251, which cannot be used in case of GF256.

9 Conclusions and Future Work

This research proposes the first hardware design of the SDitH signature scheme, a candidate in the NIST PQC addition signatures process. The results demonstrate that the signature scheme is indeed suitable for use in hardware, having many qualities that can be exploit when designed directly in hardware, such as using powers-of-two arithmetic (for GF256) and its use of parallelisable modules such as d-splitting and its natural split of signing into offline and online stages. We conclude with further work and extensions of these hardware designs and how they apply to other PQC signature schemes below.

The SDitH threshold variant Along with the SDitH NIST on-ramp signature submission, the *Threshold Variant* is another option besides the *Hypercube Variant* that provides good performance trade-offs. The main difference in the threshold variant is the way the MPC party shares are generated and verified – instead of additive sharing, the threshold variant uses Shamir secret sharing to split the plain input into polynomial evaluations (encoded as a Reed-Solomon codeword).

To adapt our hardware design to work for the threshold variant, we see that some modules requires specific reworking in order to support computing the operations inside the threshold variant. For example, the threshold variant uses a Merkle Tree to commit to the random shares, instead of using TreePRG. Therefore, a dedicated Merkle Tree builder and Merkle proof generator components are required.

However, many of the components would still work out-of-box: For example, the Key Generation routine is shared across both schemes. Moreover, due to the linearity of Shamir secret shares, we can still run pipe the data through the same ComputePlainBroadcast and PartyComputation subroutines on each party's share and obtain Shamir secret shares of the intended output. Lastly, all the modular arithmetic components we designed in this work (involving GF256 and GF251 field operations) can be shared across as well.

Applications outside of SDitH Some of the components used in our design can also be used outside of SDitH. For example, many generic MPCitH frameworks such as [KKW18], [dOT21], [BN20], and [KZ22] employ the use of seed trees (TreePRG). Hence, we can isolate the TreePRG submodule and adapt it to generate random shares for any additive secret sharing based MPCitH frameworks. The MPC computation inside SDitH is a product check, which is effectively an arithmetic circuit with a multiplication gate depth of 1. However, this is not the case when we consider other MPCitH-based signatures like Picnic or BBQ, where the MPC circuitry is more complex. With minor tweaks on ComputePlainBroadcast and PartyComputation (e.g. making them iterative and hence capable of performing multiple product checks), we can adapt the hardware design to compute more involved MPC circuitry.

[9] `https://networkbuilders.intel.com/solutionslibrary/galois-field-new-instructions-gfni-technology-guide`

Table 14: Resource and Performance comparison of our *complete* SDitH hardware design with other related PQC signature hardware designs for different security levels. †Does not include Key Generation and ‡Includes only Signature Generation.

Parameter	**FPGA Utilisation**				**Frequency**	**KeyGen**		**Sign**		**Verify**	
Sets	**LUT**	**FF**	**DSP**	**BRAM**	*(MHz)*	**Latency** *(Mcycles)*	**Time** *(ms)*	**Latency** *(Mcycles)*	**Time** *(ms)*	**Latency** *(Mcycles)*	**Time** *(ms)*
SDitH [Ours]											
SDitH-L1-GF256	16,592	8,778	0	164.5	164	0.055	0.34	6.73	41.04	8.688	52.98
SDitH-L1-GF251	17,423	16,336	196	164.5	164	0.057	0.35	21.45	130.77	23.403	142.71
SDitH-L3-GF256	22,569	13,881	0	356.0	164	0.046	0.28	7.74	47.17	10.960	66.85
SDitH-L3-GF251	29,961	25,794	382	356.0	164	0.047	0.29	24.60	149.99	24.600	149.99
SDitH-L5-GF256	23,323	14,962	0	520.5	164	0.083	0.51	17.48	106.60	24.940	152.10
SDitH-L5-GF251	34,456	31,409	472	521.5	164	0.086	0.53	45.28	276.07	30.110	183.57
PICNIC† [KRR+20]											
PICNIC-L1	90,337	23,105	0	52.5	125	–	–	0.03	0.25	0.030	0.24
PICNIC-L5	167,530	33,164	0	98.5	125	–	–	0.15	1.24	0.147	1.17
SPHINCS^{+}-simple‡ [ALC+20]											
SPHINCS^{+}-128s	48,231	72,514	0	11.5	250 & 500	–	–	–	12.40	–	0.07
SPHINCS^{+}-128f	47,991	72,505	1	11.5	250 & 500	–	–	–	1.01	–	0.16
SPHINCS^{+}-192s	48,725	72,514	0	17.0	250 & 500	–	–	–	21.40	–	0.10
SPHINCS^{+}-192f	48,398	73,476	1	17.0	250 & 500	–	–	–	1.17	–	0.19
SPHINCS^{+}-256s	51,130	74,576	1	22.5	250 & 500	–	–	–	19.30	–	0.14
SPHINCS^{+}-256f	51,009	74,539	1	22.5	250 & 500	–	–	–	2.52	–	0.21
Dilithium [ZZW+22]											
Dilithium-L2	29,998	10,336	10	11.0	97	0.004	0.04	0.03	0.29	0.004	0.05
Dilithium-L3	29,998	10,336	10	11.0	97	0.006	0.06	0.04	0.46	0.006	0.06
Dilithium-L5	29,998	10,336	10	11.0	97	0.009	0.09	0.05	0.51	0.009	0.09
LESS [BWM+23]											
LESS-L1 {b}	54,800	39,900	0	59.5	200	0.029	0.14	5.20	26.02	5.156	25.78
LESS-L1 {i}	54,800	39,900	0	59.5	200	0.077	0.38	5.13	25.63	5.093	25.47
LESS-L1 {s}	54,800	39,900	0	59.5	200	0.174	0.87	4.17	20.83	4.137	20.69
LESS-L3 {b}	76,700	57,900	0	102.5	167	0.072	0.43	39.24	234.95	39.146	234.87
LESS-L3 {s}	76,700	57,900	0	102.5	167	0.132	0.79	46.22	276.75	46.142	276.85
LESS-L5 {b}	104,300	76,700	0	167.5	143	0.134	0.93	129.89	909.20	129.726	908.08
LESS-L5 {s}	104,300	76,700	0	167.5	143	0.247	1.73	87.16	610.13	87.013	609.09

References

[AAC+22] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022 (cited on pages 1–3).

[AFG+23] C. Aguilar Melchor, T. Feneuil, N. Gama, S. Gueron, J. Howe, D. Joseph, A. Joux, E. Persichetti, T. H. Randrianarisoa, M. Rivain, and D. Yue. SDitH. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures (cited on pages 2–6, 8, 9, 11, 19, 23).

[AGH+23] C. Aguilar Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The return of the SDitH. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 564–596. Springer, Heidelberg, April 2023. DOI: 10.1007/978-3-031-30589-4_20 (cited on pages 1, 3).

[AHJ+23] C. Aguilar Melchor, A. Hülsing, D. Joseph, C. Majenz, E. Ronen, and D. Yue. SDitH in the QROM. ASIACRYPT 2023, 2023. URL: https://eprint.iacr.org/2023/756. https://eprint.iacr.org/2023/756 (cited on page 3).

[ALC+20] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden. FPGA-based SPHINCS+ implementations: Mind the glitch. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 229–237. IEEE, 2020 (cited on pages 2, 18, 20).

[AMI+22] A. Aikata, A. C. Mert, M. Imran, S. Pagliarini, and S. S. Roy. KaLi: A crystal for post-quantum security using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2022 (cited on page 2).

[BBL+18] D. J. Bernstein, L. G. Bruinderink, T. Lange, and L. Panny. HILA5 Pindakaas: on the CCA security of lattice-based encryption with error correction. In A. Joux, A. Nitaj, and T. Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 203–216. Springer, Heidelberg, May 2018. DOI: 10.1007/978-3-319-89339-6_12.

[BN20] C. Baum and A. Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 495–526. Springer, Heidelberg, May 2020. DOI: 10.1007/978-3-030-45374-9_17 (cited on page 19).

[BNG21] L. Beckwith, D. T. Nguyen, and K. Gaj. High-performance hardware implementation of CRYSTALS-Dilithium. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10. IEEE, 2021 (cited on page 2).

[BNG23] L. Beckwith, D. T. Nguyen, and K. Gaj. Hardware Accelerators for Digital Signature Algorithms Dilithium and Falcon. *IEEE Design & Test*, 2023 (cited on page 2).

[BUG+21] Q. Berthet, A. Upegui, L. Gantel, A. Duc, and G. Traverso. An area-efficient SPHINCS+ post-quantum signature coprocessor. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 180–187. IEEE, 2021 (cited on page 2).

[BWM+23] L. Beckwith, R. Wallace, K. Mohajerani, and K. Gaj. A high-performance hardware implementation of the less digital signature scheme. In T. Johansson and D. Smith-Tone, editors, *Post-Quantum Cryptography*, pages 57–90, Cham. Springer Nature Switzerland, 2023. ISBN: 978-3-031-40003-2 (cited on pages 18, 20).

[dOT21] C. de Saint Guilhem, E. Orsini, and T. Tanguy. Limbo: efficient zero-knowledge MPCitH-based arguments. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, November 2021. DOI: 10.1145/3460120.3484595 (cited on page 19).

[DXN+23] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer. Fast and efficient hardware implementation of hqc. In *Proceedings of the Selected Areas in Cryptography*, SAC, 2023 (cited on page 6).

[FJR22] T. Feneuil, A. Joux, and M. Rivain. Syndrome decoding in the head: shorter signatures from zero-knowledge proofs. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 541–572. Springer, Heidelberg, August 2022. DOI: 10.1007/978-3-031-15979-4_19 (cited on page 3).

[FR22] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022. https://eprint.iacr.org/2022/1407 (cited on page 1).

[HBD$^+$22] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 (cited on page 2).

[IKO$^+$07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In D. S. Johnson and U. Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007. DOI: 10.1145/1250790.1250794 (cited on page 3).

[KKW18] J. Katz, V. Kolesnikov, and X. Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018. DOI: 10.1145/3243734.3243805 (cited on page 19).

[KRR$^+$20] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner. Efficient FPGA implementations of LowMC and Picnic. In S. Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 417–441. Springer, Heidelberg, February 2020. DOI: 10.1007/978-3-030-40186-3_18 (cited on pages 2, 18, 20).

[KZ22] D. Kales and G. Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Report 2022/588, 2022. https://eprint.iacr.org/2022/588 (cited on page 19).

[LDK$^+$22] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 (cited on page 2).

[LSG21] G. Land, P. Sasdrich, and T. Güneysu. A hard crystal-implementing dilithium on reconfigurable hardware. In *International Conference on Smart Card Research and Advanced Applications*, pages 210–230. Springer, 2021 (cited on page 2).

[ML-DSA] FIPS 204 (Initial Public Draft): Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 204, U.S. Department of Commerce, August 2023 (cited on pages 1, 2).

[ML-KEM] FIPS 203 (Initial Public Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard. National Institute of Standards and Technology, NIST FIPS PUB 203, U.S. Department of Commerce, August 2023 (cited on page 1).

[NP17] C. Negre and T. Plantard. Efficient Regular Modular Exponentiation Using Multiplicative Half-Size Splitting. *Journal of Cryptographic Engineering*, 7(3):245–253, 2017. DOI: 10.1007/s13389-016-0134-5. URL: https://hal.archives-ouvertes.fr/hal-01185249 (cited on page 11).

[RMJ$^+$21] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias. Implementing CRYSTALS-Dilithium signature scheme on FPGAs. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–11, 2021 (cited on page 2).

[SLH-DSA] FIPS 205 (Initial Public Draft): Stateless Hash-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 205, U.S. Department of Commerce, August 2023 (cited on pages 1, 2).

[SR17] C. Sandoval-Ruiz. VHDL optimized model of a multiplier in finite fields. *Ingeniería y universidad*, 21(2):195–211, 2017 (cited on page 5).

[WZC+22] T. Wang, C. Zhang, P. Cao, and D. Gu. Efficient Implementation of Dilithium Signature Scheme on FPGA SoC Platform. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(9):1158–1171, 2022. DOI: 10.1109/TVLSI.2022.3179459 (cited on page 2).

[ZCD+20] G. Zaverucha, M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, J. Katz, X. Wang, V. Kolesnikov, and D. Kales. Picnic. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions (cited on pages 2, 3).

[ZZW+22] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu. A compact and high-performance hardware architecture for CRYSTALS-dilithium. *IACR TCHES*, 2022(1):270–295, 2022. DOI: 10.46586/tches.v2022.i1.270-295 (cited on pages 2, 18, 20).

A Appendices

A.1 Parameter Sets

Table 15: Parameters, output sizes, and performances of the SDitH signature scheme for all NIST security levels. Benchmarks use the Intel Xeon E-2378 at 2.6GHz using AVX2 from [AFG+23].

	SDitH	**NIST Security Categories**		
	Parameters	**L1**	**L3**	**L5**
Signature Parameters	NIST Security Level	143	207	272
	λ (Security Target)	128	192	256
	N^D (# Secret Shares)	2^8	2^8	2^8
	τ (Repetition Rate)	17	26	34
Syndrome Decoding	q (SD Base Field Size)	251/256	251/256	251/256
	m (Code Length)	230	352	480
	k (Vector Dimension)	126	193	278
	w (Hamming Weight Bound)	79	120	150
	d (d Splitting Size)	1	2	2
Multi-Party Computation	t (# Random evaluation points)	3	3	4
	$\mathbb{F}_q$ (SD base field)	$\mathbb{F}_q$	$\mathbb{F}_q$	$\mathbb{F}_q$
	η (Field extension size)	4	4	4
	$\mathbb{F}_{\text{points}}$ (Field extension of $\mathbb{F}_q$)	$\mathbb{F}_{q^\eta}$	$\mathbb{F}_{q^\eta}$	$\mathbb{F}_{q^\eta}$
	p (False positive probability)	$2^{-71.2}$	$2^{-72.4}$	$2^{-94.8}$
Output Sizes	pk Size (in Bytes)	120	183	234
	sk Size (in Bytes)	404	616	812
	Max Signature Size (in Bytes)	8 260	19 206	33 448

Table 16: Symmetric cryptography primitives for NIST Security Categories L1, L3, and L5 .

	NIST L1	**NIST L3**	**NIST L5**
Hash	SHA3-256	SHA3-384	SHA3-512
XOF	SHAKE-128	SHAKE-256	SHAKE-256

A.2 The SDitH Sub-Routine Procedures

For ease of reference, the list of the SDitH sub-routines have been added here.

- The ComputeQ Algorithm.
- The ComputeS equation.
- The sampling from an extendable-output function (XOF), *Sampling from XOF*.
- The sampling field elements procedure, SampleFieldElements.
- The seed expansion procedure, ExpandSeed.
- The expand MPC challenge procedure, ExpandMPCChallenge.
- The expand of the view-opening challenge procedure, ExpandViewChallenge.
- The get seed sibling path procedure, GetSeedSiblingPath.
- The commitments procedure, Commit.
- The polynomial evaluation procedure, Evaluate.

ComputeQ

Algorithm 1 Pseudocode for the Shift-and-Multiply algorithm used for ComputeQ.

```
Input:  pos[w/d]
Output: q[w/d+1]
#initialisation
q[0] = 1
for i in range (1,w/d+1):
    q[i] = 0
#shift and multiply
for i in range (1,w/d+1):
  for j in range(i + 1, j >= 1, j-1)
    q[j] = q[j-1] + (q[j]*pos[i])
  q[0] = q[0]*pos[i]
```

ComputeS

$$S(x) = \sum_{i=1}^{m/d}(\prod_{j\in[1:m/d]} f_i - f_j)^{-1})(x_i)(\prod_{j\in[1:m/d]} (f_i - f_j)^{-1})(\prod_{i=1}^{m/d}(X - f_i))/(X - f_i) \quad (1)$$

Sampling from XOF. We shall denote by Sample, the routine generating pseudorandom element from an arbitrary set $\mathcal{V}$. A call to

$$v \quad \text{XOF.Sample}(\mathcal{V})$$

outputs a uniform random element $v \in \mathcal{V}$. The Sample routine relies on calls to GetByte to generate pseudorandom bytes which are then formatted to obtain a uniform variable $v \in \mathcal{V}$, possibly using rejection sampling. The implementation of Sample depends on the target set $\mathcal{V}$. We detail the case of sampling field elements hereafter, namely when $\mathcal{V} = \mathbb{F}_q^n$ for some n.

Sampling field elements. The subroutine XOF. SampleFieldElements(n) samples n pseudorandom elements from $\mathbb{F}_q$. It assumes that the XOF has been previously initialised by a call to XOF.Init($\cdot$). The implementation of the SampleFieldElements routine use the following process. It first generates a stream of bytes $B_1, \ldots, B_{n'}$ for some $n' \geq n$. Those bytes are converted into n field elements as follows:

- For $\mathbb{F}_q = \mathbb{F}_{256}$: The byte B_i is simply returned as the ith sampled field element. The XOF is called to generate $n' = n$ bytes.
- For $\mathbb{F}_q = \mathbb{F}_{251}$: The byte B_i is interpreted as an integer $B_i \in \{0, 1, \ldots, 255\}$. We use the principle of rejection sampling to only select integer values modulo 251, namely we reject byte values in $\{251, \ldots, 255\}$. The procedure goes as follows:

```
1: $i = 1$
2: while $i \leq n$ do
3:     $B$    XOF.GetByte()
4:     if $B \in \{0, 1, \ldots, 250\}$ then
5:         $f_i = B$; $i$ ++
6: return $(f_1, \ldots, f_n)$
```

  The number of generated bytes n' which are necessary to complete the process is non-deterministic. In average on needs to generates $n' \approx (256/251)n \approx 1.02n$ bytes.

Seed expansion. The subroutine ExpandSeed expands a salt and a master seed into a given number of seeds. Specifically, a call to ExpandSeed($\mathsf{salt}, \mathsf{seed}, n$) initialises the XOF by calling XOF.Init($\mathsf{salt} \parallel \mathsf{seed}$) and then calls XOF.GetByte() to generate a stream of bytes $B_1, \ldots, B_{n\lambda/8}$ which are divided into n output λ-bit seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_n$ as follows:

$$(\underbrace{B_1, \ldots, B_{\lambda/8}}_{\mathsf{seed}_1}, \ldots, \underbrace{B_{(n-1)\lambda/8+1}, \ldots, B_{n\lambda/8}}_{\mathsf{seed}_n})$$

Expansion of MPC challenge. The subroutine ExpandMPCChallenge expands the first Fiat-Shamir hash h_1 into the MPC challenges $(r, \boldsymbol{\varepsilon}) \in \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$. This subroutine takes as input the hash h_1 and the number n of pairs $(r, \boldsymbol{\varepsilon})$ to be generated. It consists of the following steps:

$$\begin{aligned}
&\text{XOF.Init}(h_1) \\
&v \quad \text{XOF. SampleFieldElements}(nt\eta(d+1)) \\
&(\mathsf{chal}\,\texttt{[}1\texttt{]}, \ldots, \mathsf{chal}\,\texttt{[}n\texttt{]}) = \text{Parse}(v, \mathbb{F}_q^{t\eta(d+1)}, \ldots, \mathbb{F}_q^{t\eta(d+1)}) \ ,
\end{aligned}$$

where each $\mathsf{chal}\,\texttt{[}e\texttt{]}$ represents a serialised pair $(r, \boldsymbol{\varepsilon}) \in \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$.

For the hypercube variant we have one challenge per parallel execution, *i.e.* $n = \tau$, while for the threshold variant, we use a global challenge for all the executions, *i.e.* $n = 1$.

Expansion of view-opening challenge. The subroutine ExpandViewChallenge, expands the second Fiat-Shamir hash h_2 into the view-opening challenge $I\,\texttt{[}1\texttt{]}, \ldots, I\,\texttt{[}\tau\texttt{]}$, where $I\,\texttt{[}e\texttt{]} \subset [1:N]$ is the set of parties to be opened for execution e. This subroutine takes as input the hash h_2 and a mode character, either `hypercube` or `threshold`. It first initialises the XOF by calling

$$\text{XOF.Init}(h_2) \ .$$

For the `hypercube` mode, the generated sets are of cardinal $N - 1$ and are simply represented by the indexes $i^*\,\texttt{[}1\texttt{]}, \ldots, i^*\,\texttt{[}\tau\texttt{]}$ such that $I\,\texttt{[}e\texttt{]} = [1:N] \setminus i^*\,\texttt{[}e\texttt{]}$ for each execution e. The subroutine then calls

$$i^*[e] \quad \text{XOF.Sample}([1:N]) \quad \forall e \in [1:\tau] \ .$$

For the `threshold` mode, the generated sets are of cardinal ℓ. The subroutine then calls

$$I[e] \quad \text{XOF.Sample}(\{J \subseteq [1:N] \ ; \ |J| = \ell\}) \quad \forall e \in [1:\tau] \ .$$

GetSeedSiblingPath This subroutine takes a 2λ-bit salt, a λ-bit seed and an index i^*, and it returns the sibling path of the seed leave indexed by i^* in a binary seed tree. It returns the D seeds that are sibling of the ancestors of i^* in the tree, namely:

$$\mathsf{path}_j = \mathsf{seed}_{(i^* \gg (D-j)) \oplus 1} \text{ for } j \in [1, D]$$

Here, $\gg$ is the logical right shift, and $\oplus 1$ flips the least significant bit. It is possible to store the $2.2^D - 1$ seeds, extract the sibling path from it, and delete the remaining seeds, or equivalently to re-derive those seeds from the root seed and the salt in D calls of the derivation formula above.

Commitments. The subroutine Commit takes as input a 2λ-bit salt, an execution index e, a share index i and some data $\mathsf{data} \in \{0,1\}^*$. It hashes them all together and returns the corresponding digest. Specifically, we define:

$$\text{Commit}(\mathsf{salt}, e, i, \mathsf{data}) = \text{Hash}(0 \parallel \mathsf{salt} \parallel e_0 \parallel e_1 \parallel i_0 \parallel i_1 \parallel \mathsf{data}) \ ,$$

where e_0, e_1, i_0, i_1 are the byte values such that $e = e_0 + 256 \cdot e_1$ and $i = i_0 = 256 \cdot i_1$, where 0, e_0, e_1, i_0 and i_1 are encoded on one byte, and where salt is encoded on $2\lambda/8$ bytes.

Polynomial evaluation. We define the function Evaluate which takes as input an $\mathbb{F}_q$-vector Q representing the coefficients of polynomial of $\mathbb{F}_q[X]$ and a point $r \in \mathbb{F}_{\text{points}}$, computes the evaluation $Q(r)$. Formally, we have

$$\text{Evaluate} : \begin{cases} \bigcup_{|Q|} (\mathbb{F}_q)^{|Q|} \times \mathbb{F}_{q^\eta} & \to\leftarrow \mathbb{F}_{q^\eta} \\ (Q, r) & \mapsto\leftarrow \sum_{i=1}^{|Q|} Q\,[i] \cdot r^{i-1} \end{cases} \qquad \text{where } r^{i-1} = \underbrace{r \quad r \quad \cdots \leftarrow r}_{i-1 \text{ times}} \ .$$

Let us stress that the powers r^i lies on the extension field $\mathbb{F}_{q^\eta}$ while the polynomial coefficients $Q\,[i+1]$ lies on the base field $\mathbb{F}_q$.

A.3 The SDitH Subroutine Algorithms

- Algorithm 2 for ComputePlainBroadcast.
- Algorithm 3 for PartyComputation.
- Algorithm 4 for SampleWitness.

Algorithm 2 ComputePlainBroadcast

```
Input: input_plain := (wit_plain, beav_ab_plain, beav_c_plain), chal, $(H', y)$
Output: broad_plain
 1: $(s_A, \boldsymbol{Q'}, \boldsymbol{P})$   Parse(wit_plain, $\mathbb{F}_q^k, (\mathbb{F}_q^{w/d})^d, (\mathbb{F}_q^{w/d})^d$)
 2: $(\boldsymbol{a}, \boldsymbol{b})$   Parse(beav_ab_plain, $(\mathbb{F}_{q^\eta}^d)^t$)
 3: $c$   Parse(beav_c_plain, $\mathbb{F}_{q^\eta}^t$)
 4: $(r, \boldsymbol{\varepsilon})$   Parse(chal, $\mathbb{F}_{q^\eta}^t, (\mathbb{F}_{q^\eta}^d)^t$)
 5: $s = (s_A \mid y + H' s_A)$
 6: $\boldsymbol{Q}$ = CompleteQ($\boldsymbol{Q'}$, 1)                                ▷ See Section 3.3
 7: $\boldsymbol{S}$   Parse($s, (\mathbb{F}_q^{m/d})^d$)
 8: for $j \in [1:t]$ do
 9:     for $\nu \in [1:d]$ do
10:         $\boldsymbol{\alpha}\,[j]\,[\nu] = \boldsymbol{\varepsilon}\,[j]\,[\nu]$   Evaluate($\boldsymbol{Q}\,[\nu], r\,[j]$) + $\boldsymbol{a}\,[j]\,[\nu]$     ▷ See Section 4.1
11:         $\boldsymbol{\beta}\,[j]\,[\nu]$ = Evaluate($\boldsymbol{S}\,[\nu], r\,[j]$) + $\boldsymbol{b}\,[j]\,[\nu]$      ▷ See Section 4.1
12: broad_plain = Serialize($\boldsymbol{\alpha}, \boldsymbol{\beta}$)
13: return broad_plain
```

Algorithm 3 PartyComputation

```
Input: input_share := (wit_share, beav_ab_share, beav_c_share), chal, $(H', y)$, broad_plain, with_offset
Output: broad_share
 1: $(s_A, \boldsymbol{Q'}, \boldsymbol{P})$    Parse(wit_share, $\mathbb{F}_q^k$, $(\mathbb{F}_q^{w/d})^d$, $(\mathbb{F}_q^{w/d})^d$)
 2: $(\boldsymbol{a}, \boldsymbol{b})$    Parse(beav_ab_share, $(\mathbb{F}_{q^\eta}^d)^t$)
 3: $c$    Parse(beav_c_share, $\mathbb{F}_{q^\eta}^t$)
 4: $(r, \boldsymbol{\varepsilon})$    Parse(chal, $\mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$)
 5: $(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\beta}})$    Parse(broad_plain, $(\mathbb{F}_{q^\eta}^d)^t$, $(\mathbb{F}_{q^\eta}^d)^t$)
 6: if with_offset is True then
 7:     $s = (s_A \mid y + H's_A)$
 8:     $\boldsymbol{Q} = (\boldsymbol{Q'}, 1)$
 9: else
10:     $s = (s_A \mid H's_A)$
11:     $\boldsymbol{Q} = (\boldsymbol{Q'}, 0)$
12: $\boldsymbol{S}$    Parse($s$, $(\mathbb{F}_q^{m/d})^d$)
13: for $j \in [1 : t]$ do
14:     $v[j] = -c[j]$
15:     for $\nu \in [1 : d]$ do
16:         $\boldsymbol{\alpha}[j][\nu] = \boldsymbol{\varepsilon}[j][\nu]$    Evaluate($\boldsymbol{Q}[\nu], r[j]$) + $\boldsymbol{a}[j][\nu]$        ▷ See Section 4.1
17:         $\boldsymbol{\beta}[j][\nu]$ = Evaluate($\boldsymbol{S}[\nu], r[j]$) + $\boldsymbol{b}[j][\nu]$        ▷ See Section 4.1
18:         $v[j]$ += $\boldsymbol{\varepsilon}[j][\nu]$    Evaluate($F, r[j]$)    Evaluate($\boldsymbol{P}[\nu], r[j]$)        ▷ See Section 4.1
19:         $v[j]$ += $\bar{\boldsymbol{\alpha}}[j][\nu]$    $\boldsymbol{b}[j][\nu] + \bar{\boldsymbol{\beta}}[j][\nu]$    $\boldsymbol{a}[j][\nu]$
20:         if with_offset is True then
21:             $v[j]$ += $-\boldsymbol{\alpha}[j][\nu]$    $\boldsymbol{\beta}[j][\nu]$
22: broad_share = Serialize($\boldsymbol{\alpha}, \boldsymbol{\beta}, v$)
23: return broad_share
```

Algorithm 4 SampleWitness

```
Input: $\text{seed}_{\text{wit}} \in \{0,1\}^\lambda$
Output: $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$
 1: $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$    Init($(\mathbb{F}_q^{w/d+1})^d$, $(\mathbb{F}_q^{m/d})^d$, $(\mathbb{F}_q^{w/d})^d$)
 2: XOF.Init($\text{seed}_{\text{wit}}$)
 3: for $\nu \in [1 : d]$ do
 4:     pos[$\nu$]    XOF.Sample($\{J \subseteq [1 : m/d]\ ;\ |J| = w/d\}$)
 5:     val[$\nu$]    XOF.Sample($(\mathbb{F}_q^*)^{w/d}$)
 6:     for $i \in [1 : m/d]$ do
 7:         $\boldsymbol{x}[\nu][i] = \sum_{j \in [1:w/d]}$ val[$\nu$][$j$] $\cdot$ (pos[$\nu$][$j$]==$i$)
 8:
 9:     $\boldsymbol{Q}[\nu]$ = ComputeQ(pos[$\nu$])        ▷ See Section 3.3
10:     $\boldsymbol{S}[\nu]$ = ComputeS($\boldsymbol{x}[\nu]$)        ▷ See Section 3.3
11:     $\boldsymbol{P}[\nu]$ = ComputeP($\boldsymbol{Q}[\nu], \boldsymbol{S}[\nu]$)        ▷ See Section 3.3
12: return $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$
```

A.4 Key Generation, Signature Generation, and Signature Verification Algorithms

- Algorithm 5 for KeyGen.
- Algorithm 6a for Sign Offline.
- Algorithm 6b for Sign Online.
- Algorithm 7 for Verify.

Algorithm 5 SDitH – Key Generation

```
1:  $\mathsf{seed}_{\mathsf{root}}$    $\{0,1\}^\lambda$
2:  $(\mathsf{seed}_{\mathsf{wit}}, \mathsf{seed}_H)$    $\mathrm{ExpandSeed}(\mathsf{salt} := 0, \mathsf{seed}_{\mathsf{root}}, 2)$    ▷ See Section 3.2.2
3:  $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$    $\mathrm{SampleWitness}(\mathsf{seed}_{\mathsf{wit}})$    ▷ See Section 3.3
4:  $s = \mathrm{Serialize}(\boldsymbol{S})$
5:  $(s_A, s_B) = \mathrm{Parse}(s, \mathbb{F}_q^k, \mathbb{F}_q^{m-k})$
6:  $H'$    $\mathrm{ExpandH}(\mathsf{seed}_H)$
7:  $y = s_B + H' s_A$    ▷ See Section 3.4
8:  $\boldsymbol{Q'} = \mathrm{TruncateQ}(\boldsymbol{Q})$
9:  $\mathsf{wit\_plain} = \mathrm{Serialize}(s_A, \boldsymbol{Q'}, \boldsymbol{P})$
10: return $(pk = (\mathsf{seed}_H, y), sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain}))$
```

Algorithm 6a SDitH – Hypercube Variant – Signature Generation (Offline Part)

```
Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$
1:  $\mathsf{salt}$    $\{0,1\}^{2\lambda}$, $\mathsf{mseed}$    $\{0,1\}^\lambda$
2:  $H'$    $\mathrm{ExpandH}(\mathsf{seed}_H)$
3:  $\{\mathsf{rseed}[e]\}_{e \in [1:\tau]}$    $\mathrm{ExpandSeed}(\mathsf{salt}, \mathsf{mseed}, \tau)$    ▷ See Section 3.2.2
4:  for $e \in [1:\tau]$ do
5:      $(\mathsf{seed}[e][i])_{i \in [1:2^D]}$    $\mathrm{TreePRG}(\mathsf{salt}, \mathsf{rseed}[e])$    ▷ See Section 3.2.9
6:      $\mathsf{acc} = 0$
7:      $\mathsf{input\_mshare}[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
8:      for $i \in [1:2^D]$ do
9:          if $i \neq 2^D$ then
10:             $\mathsf{input\_share}[e][i]$    $\mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$
11:                                                                  ▷ See Section 3.2.1
12:             $\mathsf{acc}$ += $\mathsf{input\_share}[e][i]$
13:             $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
14:             for $p \in [1:D]$ : the $p^{\text{th}}$ bit of $i-1$ is zero, do
15:                 $\mathsf{input\_mshare}[e][p]$ += $\mathsf{input\_share}[e][i]$
16:         else
17:             $\mathsf{acc\_wit}, \mathsf{acc\_beav\_ab}, \mathsf{acc\_beav\_c} = \mathsf{acc}$
18:             $\mathsf{beav\_ab\_plain}[e] = \mathsf{acc\_beav\_ab} + \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], 2dt\eta)$
19:                                                                  ▷ See Section 3.2.1
20:             $\mathsf{beav\_c\_plain}[e] = \mathsf{beav\_c\_plain}$    $\mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$
21:             $\mathsf{aux}[e] = (\mathsf{wit\_plain} - \mathsf{acc\_wit}, \mathsf{beav\_c\_plain}[e] - \mathsf{acc\_beav\_c})$
22:             $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
23:         $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$    ▷ See Section 3.2.7
24: $h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$    ▷ See Section 3.2.8
25: $(\mathsf{chal}[e])_{e \in [1:\tau]}$    $\mathrm{ExpandMPCChallenge}(h_1, \tau)$    ▷ See Section 3.2.3
26: for $e \in [1:\tau]$ do
27:     $\mathsf{input\_plain}[e] = (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}[e], \mathsf{beav\_c\_plain}[e])$
28:     $\mathsf{broad\_plain}[e]$    $\mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}[e], \mathsf{chal}[e], (H', y))$    ▷ See Section 4.2
```

Algorithm 6b SDitH – Hypercube Variant – Signature Generation (Online Part)

```
29: for $e \in [1 : \tau]$ do
30:     for $p \in [1 : D]$ do
31:         $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}[e][p], \mathsf{chal}[e],$      ▷ See Section 4.3
32:                                                  $(H', y), \mathsf{broad\_plain}[e], \texttt{False})$
33: $h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$      ▷ See Section 3.2.8
34: $\{i^*[e]\}_{e\in[1:\tau]}$     $\mathrm{ExpandViewChallenge}(h_2, 1)$
35: for $e \in [1 : \tau]$ do
36:     $\mathsf{path}[e]$     $\mathrm{GetSeedSiblingPath}(\mathsf{rseed}[e], i^*[e])$      ▷ See Section 3.2.5
37:     if $i^*[e] = 2^D$ then
38:         $\mathsf{view}[e] = \mathsf{path}[e]$
39:     else
40:         $\mathsf{view}[e] = (\mathsf{path}[e], \mathsf{aux}[e])$
41: return $\sigma = \Big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\Big)$
```

Algorithm 7 SDitH – Hypercube Variant – Verification Algorithm

Input: a public key $pk = (\mathsf{seed}_H, y)$, a signature σ and a message $m \in \{0,1\}^*$

```
 1: Parse $\sigma$ as $\Big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\Big)$
 2: $H'$     $\mathrm{ExpandH}(\mathsf{seed}_H)$
 3: $\{i^*[e]\}_{e\in[1:\tau]}$     $\mathrm{ExpandViewChallenge}(h_2, 1)$      ▷ See Section 3.2.4
 4: for $e \in [1 : \tau]$ do
 5:     $(\mathsf{seed}[e][i])_{i\in[1:2^D\setminus i^*[e]]}$     $\mathrm{GetLeavesFromSiblingPath}(i^*[e], \mathsf{salt}, \mathsf{path}[e])$      ▷ See Section 3.2.6
 6:     for $i \in \{2^D \setminus i^*[e]\}$ do
 7:         if $i \neq 2^D$ then
 8:             $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
 9:         else
10:             $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
11:         $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$      ▷ See Section 3.2.7
12: $h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$      ▷ See Section 3.2.8
13: $\mathsf{chal}$     $\mathrm{ExpandMPCChallenge}(h_1, \tau)$      ▷ See Section 3.2.3
14: for $e \in [1 : \tau]$ do
15:     $\mathsf{input\_mshare}^*[e][p] = 0$ for all $(e, p) \in [1 : \tau] \times [1 : D]$
16:     for $i \in [1 : 2^D \setminus i^*[e]]$ do
17:         if $i \neq 2^D$ then
18:             $\mathsf{input\_share}[e][i]$     $\mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$
19:                                                  ▷ See Section 3.2.1
20:         else
21:             $\mathsf{beav\_ab\_plain}[e][2^D] = \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][2^D], 2dt\eta)$
22:                                                  ▷ See Section 3.2.1
23:             $\mathsf{input\_share}[e][2^D] = (\mathsf{aux}[e] \mid \mathsf{beav\_ab\_plain}[e][2^D])$
24:         for $p \in [1 : D]$ : the $p^{\text{th}}$ bit of $i - 1$ and $i^*[e]$ are different do
25:             $\mathsf{input\_mshare}'[e][p]$ += $\mathsf{input\_share}[e][i]$
26:     for $p \in [1 : D]$ do
27:         if the $p^{\text{th}}$ bit of $i^*[e]$ is 1 then
28:             $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal},$      ▷ See Section 4.3
29:                                                  $(H', y), \mathsf{broad\_plain}, \texttt{False})$
30:         else
31:             $\mathsf{broad\_share}[e][p] = \mathsf{broad\_plain}[e] - \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal},$
32:                                                  $(H', y), \mathsf{broad\_plain}, \texttt{True})$
33:                                                  ▷ See Section 4.3
34: $h_2' = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.      ▷ See Section 3.2.8
35: return $h_2 \overset{?}{=} h_2'$
```

A.5 Additional Figures

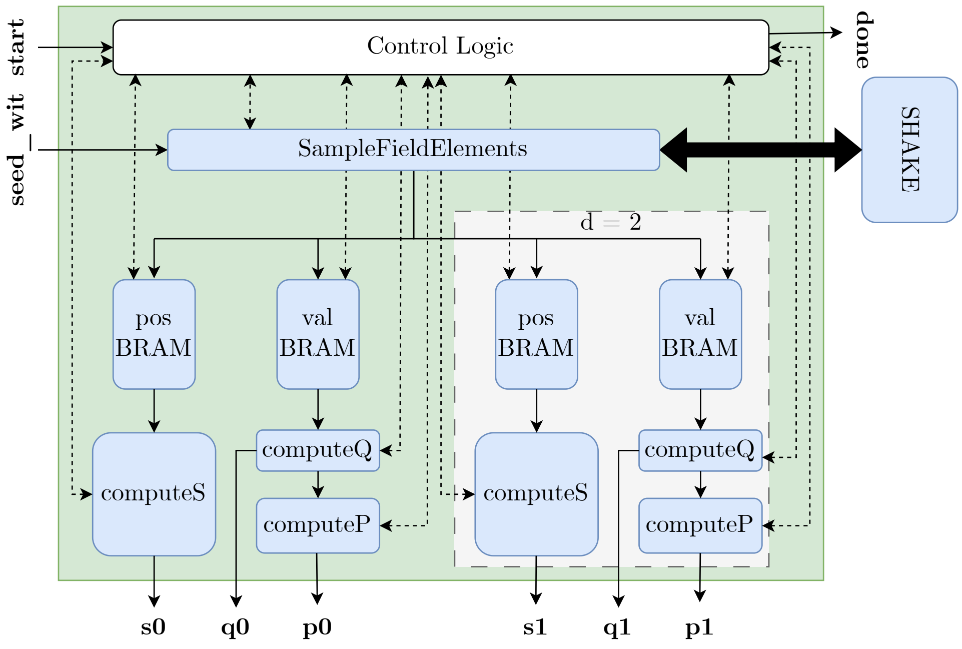

Figure 5: Hardware block design for the SampleWitness Module. The greyed part shows that this part is enabled at compile time, only for security levels L3 and L5, where $d = 2$.

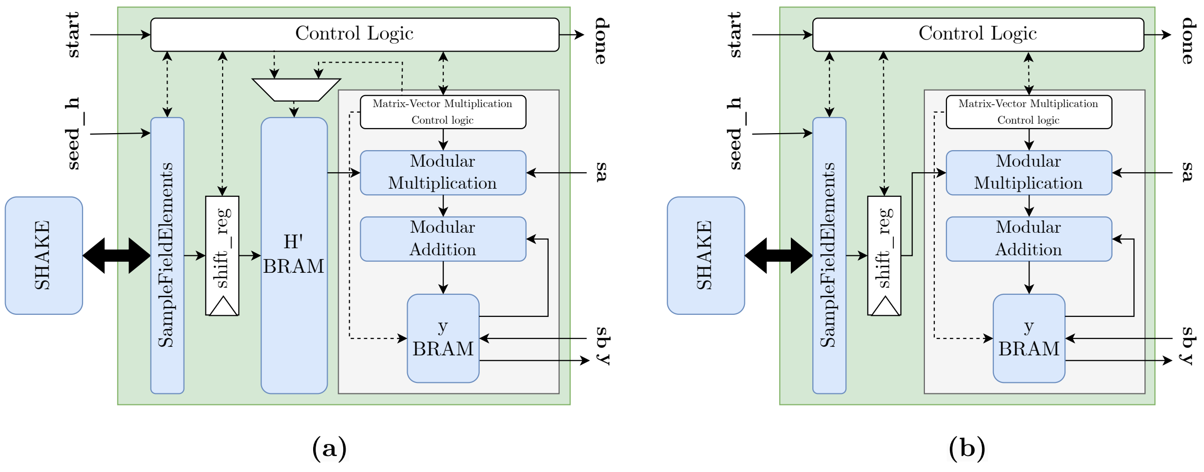

Figure 6: Hardware designs for Syndrome Decoding Instance Module interfaced with the SHAKE module using (a) Sample First and then Multiply (SFTM) and (b) Sample and Multiple On-the-fly (SaMO).

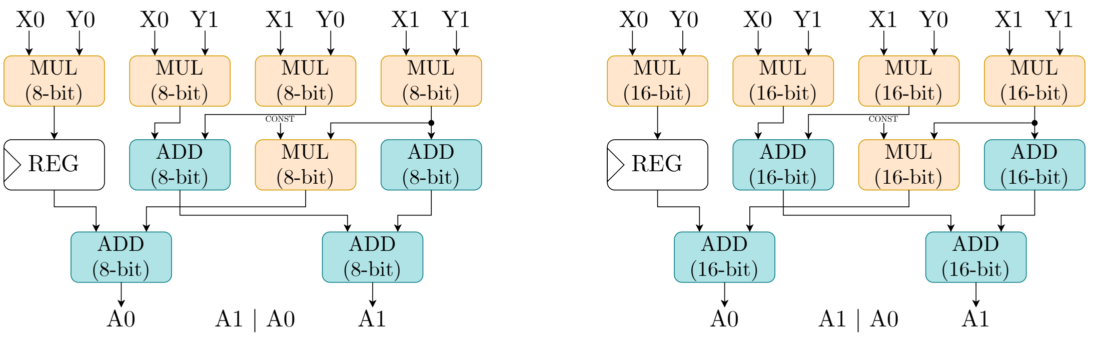

Figure 7: (a) 16-bit multiplications constructed from 8-bit multiplications and additions and (b) 32-bit multiplications constructed from 16-bit multiplications and additions. In both (a) and (b) X0 Y0 A0 represents lower 8 or 16 bits and X1 Y1 A1 represents high 8 or 16 bits.

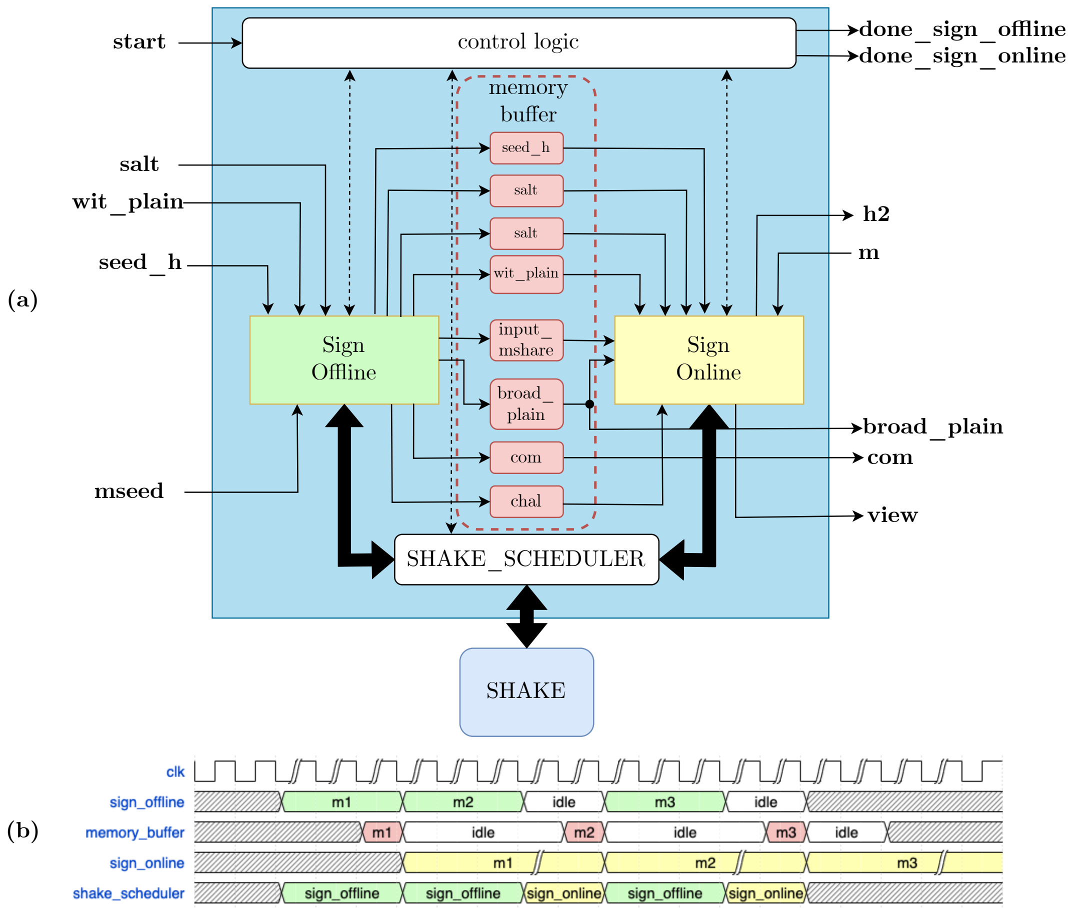

Figure 8: (a) Hardware block design of the full signature generation module where the offline and online phases are working in tandem. (b) The timing diagram showing how the offline and online phases would work in an interleaved fashion while signing multiple messages.