SDitH in Hardware

Sanjay Deshpande[1,2], James Howe[1], Jakub Szefer[2] and Dongze Yue[1]

[1] SandboxAQ, Palo Alto, USA, firstname.lastname@sandboxaq.com
[2] Yale University, New Haven, USA, firstname.lastname@yale.edu

Abstract. This work presents the first hardware realisation of the Syndrome-Decoding-in-the-Head (SDitH) signature scheme, which is a candidate in the NIST PQC process for standardising post-quantum secure digital signature schemes. SDitH's hardness is based on conservative code-based assumptions, and it uses the Multi-Party-Computation-in-the-Head (MPCitH) construction. This is the first hardware design of a code-based signature scheme based on traditional decoding problems and only the second for MPCitH constructions, after Picnic. This work presents optimised designs to achieve the best area efficiency, which we evaluate using the Time-Area Product (TAP) metric. This work also proposes a novel hardware architecture by dividing the signature generation algorithm into two phases, namely offline and online phases for optimising the overall clock cycle count. The hardware designs for key generation, signature generation, and signature verification are parameterised for all SDitH parameters, including the NIST security levels, both syndrome decoding base fields (GF256 and GF251), and thus conforms to the SDitH specifications. The hardware design further supports secret share splitting, and the hypercube optimisation which can be applied in this and multiple other NIST PQC candidates. 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. Our key generation outperforms software drastically, giving a 11-17x reduction in *runtime*, despite the significantly faster clock speed. On Artix 7 FPGAs we can perform key generation in 55.1 Kcycles, signature generation in 6.7 Mcycles, and signature verification in 8.6 Mcycles for NIST L1 parameters, which increase for GF251, and for L3 and L5 parameters.

Keywords: Hardware Security · NIST PQC · FPGA · Post-Quantum cryptography · SDitH

1 Introduction

In 2022, NIST selected the first set of post-quantum cryptography (PQC) standards [AAC+22] intended to replace our current public-key cryptography standards (RSA and ECC) which are vulnerable to quantum attacks run on a cryptographically relevant fault-tolerant quantum computer. These new standards, which are designed to be secure against both classical and quantum computers, are known as ML-KEM [ML-KEM] (aka CRYSTALS-Kyber), ML-DSA [ML-DSA] (aka CRYSTALS-Dilithium), SLH-DSA [SLH-DSA] (aka SPHINCS$^+$), and FN-DSA (aka Falcon). The former is the only key encapsulation mechanism (KEM) standard, and the latter three are each a digital signature algorithm (DSA). The selected candidates, however, have limited diversity in terms of their hardness assumptions. As result, NIST has requested further candidates to be submitted, among which SDitH is one of them. Evaluating SDitH, and others, in hardware is critical to help understand how efficient they can be, which is one of practical considerations that NIST must account for in selecting these new additional PQC signatures.

In June 2023, NIST announced this additional call[1] for additional PQC signature submissions to complement and add diversity to Dilithium, SPHINCS$^+$, and Falcon, the current PQC signature standards, and in July 2023, they accepted 40 new proposals[2]. Out of 40 submissions, 7 of these used MPCitH, with one of these being SDitH. The SDitH scheme comes with two variants, a hypercube version – based on [AGH+23] – and a threshold version – based on [FR22], as well as offering parameters for two finite fields GF256 and GF251. These two variants and parameter sets offer a variety of signature sizes and performance profiles; in this work we will concentrate on the hypercube variant.

The SDitH signature scheme is a relatively new proposal, with this research being the first presentation of its design in hardware. SDitH is based on conservative code-based hardness assumptions and utilises

[1] https://csrc.nist.gov/News/2023/additional-pqc-digital-signature-candidates.
[2] https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

the MPCitH paradigm. Since SDitH is a candidate in the NIST PQC process for additional signatures, this work helps to establish a basis upon which it can be compared to the current NIST PQC signature standards which have hardware design. Existing standardised algorithms with hardware implementations include ML-DSA [ML-DSA] (aka CRYSTALS-Dilithium [LDK+22]) and SLH-DSA [SLH-DSA] (aka SPHINCS$^+$ [HBD+22]). Additionally, the NIST PQC candidate Picnic [ZCD+20], which was eventually removed from consideration by NIST after round 3 [AAC+22], is also relevant to compare to since it is also an MPCitH-based signature scheme.

To-date, researchers working on hardware designs have heavily focused on Dilithium and they have presented designs ranging from designs targeting high-throughput to low-cost [LSG21; RMJ+21; BNG21; ZZW+22; AMI+22; WZC+22; BNG23]. There have also been some hardware designs for SPHINCS$^+$ [ALC+20; BUG+21] and Picnic [KRR+20]. Given this pool of hardware designs, it is naturally critical for any new scheme such as SDitH to also develop hardware, and evaluate it against the other schemes.

For SDitH evaluation and comparisons we focus on analysing SDitH with respect to (i) SPHINCS$^+$ since it is closer in terms of conservatism, performance, and signature size, (ii) Picnic, due to the shared MPCitH basis, (iii) Dilithium, since it has many hardware designs and it is the main PQC signature standard, and lastly (iv) the optimised AVX2 software implementation of SDitH presented in the specifications [AFG+23] which use a Intel Xeon E-2378 running at 2.6 GHz. This is also the first implementation of any of the MPCitH-based candidates that were submitted to NIST in the latest call for additional signature schemes. In order to easily compare with the other NIST candidates that have been implemented in hardware[3], this work utilises Artix-7 FPGAs for all evaluation.

1.1 Contributions

The contributions of this work are as follows:

1. The first hardware design of the SDitH signature scheme for the hypercube variant, and using all proposed parameter sets. All hardware designs are specification compliant, constant-time[4], and also parameterisable in terms of the security level (λ), syndrome decoding field size (q), share splitting size (d), the repetition rate (τ), and the random evaluation points (t) parameters.

2. We design an optimised sample and matrix-vector multiplication core, `syndrome_decoding`, for use in key generation, signature generation, and signature verification. It also includes two design variations, sample-first-then-multiply (SFTM) and sample-and-multiply-on-the-fly (SaMO).

3. We exploit the nature of the SDitH signature generation design by providing a parametrisable option to split signature generation into two stages, namely *offline* and *online*, in order to hide many of the clock cycles required. Through this we reduce clock cycles by 27-33% with an additional cost of 30%-60% in BRAMs.

4. While we design separate hardware modules for key generation, signature generation, and signature verification operations, we provide the capability that they all could be combined into one design consisting of all operations together.

5. For NIST security levels L3 and L5, we take advantage of d-splitting size syndrome decoding parameter, which adds more parametrisable options and allows the scope of additional parallelism, specifically in SampleWitness, ComputePlainBroadcast, and PartyComputation modules.

6. Many of the sub-modules designed could also be useful in other MPCitH-based schemes or those which employ the hypercube optimisation.

7. We evaluate the resource requirements of our hardware designs on a Xilinx Artix-7 (`xc7a200t`) FPGA, as recommended by NIST for the PQC standardisation process.

8. The hardware code is open-source and released under an Apache-2.0 licence, available at:
   https://github.com/sandbox-quantum/sdith-impl-hw.

[3] NIST PQC Standardization Update - Round 2 and Beyond, https://csrc.nist.gov/CSRC/media/Presentations/pqc-update-round-2-and-beyond/images-media/pqcrypto-sept2020-moody.pdf.

[4] The randomness sampling (SampleFieldElements) module has variable runtime, but this only affects public information. This conforms to the specification and the reference implementation. We elaborate on this throughout the paper.

2 Preliminaries

The Syndrome-Decoding-in-the-Head (SDitH) cryptosystem [AFG+23] is a digital signature scheme based on the hardness of the Syndrome Decoding (SD) problem for random linear codes on a finite field. At its core, the signature scheme is a zero-knowledge proof of knowledge of a low-weight solution, x, of a SD instance $y = Hx$. The Fiat-Shamir transform is also used for the signature to allow for non-interactivity.

The construction of the zero-knowledge proof is made distinctly more efficient by building the Multi-Party-Computation-in-the-Head (MPCitH) [IKO+07] construction into the protocol. This is similar to a previous candidate to the NIST PQC process called Picnic [ZCD+20], which was not finally chosen as a PQC signature standard due to NIST preferring SPHINCS$^+$, the decision in part was motivated by the expectation that "future cryptosystems that evolve out of the multi-party-computation-in-the-head paradigm may eventually prove significantly superior to the third-round Picnic design" [AAC+22]. The conservative assumptions and very good performance profiles of SDitH make a natural challenger in the category of schemes leveraging MPCitH.

2.1 The SDitH Cryptosystem

This section describes the SDitH key generation, signature generation, and signature verification algorithms in Algorithm 1, Algorithm 2a and Algorithm 2b, and Algorithm 7, respectively. In the algorithm description, we add links to each section of this paper describing the relevant portion of the hardware designs. For the original algorithm specifications and details of SDitH, we refer the reader to the specification document [AFG+23].

SDitH Key Generation The SDitH key generation procedure is shown in Algorithm 1. The procedure effectively consists of sampling a random SD instance, $y = Hx$, where the public-key is the instance (H, y) and the secret-key is the low-weight solution to this, x, which also contains the public elements, and is thus (H, y, x). The key sizes are made smaller by outputting the seeds (e.g., seed_H) and re-generating data from the seeds, e.g., H, in the sign and verify procedures.

After profiling the operations in key generation, the SampleWitness module would be a potential bottleneck for clock cycles, which internally consists of three main operations used in Algorithm 6, ComputeQ, ComputeP, and ComputeS. In our hardware implementation, we take advantage of hardware parallelism and other efficient implementation tactics to schedule these operations to run in parallel (described in Section 3.3). Due to these optimisations, we show later (in Table 17) that our hardware design outperforms the software implementation by a significant margin.

Algorithm 1 SDitH – Key Generation

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

SDitH Signature Generation The high-level idea behind the signature generation procedure is to simulate a number (τ) of MPCitH instances in order to show that the signer (i.e. prover) has (i) a low-weight solution to $y = H'x$ and (ii) that they did a zero-knowledge proof without cheating, to the verifier.

The two interactions in the simulation between the prover and verifier make this a 5-round[5] honest-verifier zero-knowledge (HVZK) interactive protocol, which is then converted into a non-interactive

[5] It is worth noting that in the QROM proof [AHJ+23], this 5-round protocol was represented as a 3-round protocol.

signature scheme via the Fiat-Shamir transform. The reason the process is repeated τ times is to amplify the soundness (ε) of the procedure to reach the desired security goal, i.e., so that $\varepsilon^{\tau} \leq 2^{-\lambda}$.

This soundness was at the core of the Hypercube-MPCitH approach [AGH+23], which was used to improve the original SDitH proposal [FJR22]. The Hypercube-MPCitH approach amplifies the soundness of any MPC protocol that uses additive secret sharing by taking N^D shares and compounding them into a hypercube of dimension D. This affects the procedure by increasing soundness from $1/N$ to $1/N^D$, meaning we require fewer repetitions which in turn produces a smaller signature. This means we require N^D *offline* computations in (i) but as a result only $N \times D$ *online* computations in (ii). The Hypercube-MPCitH approach was subsequently used in 3 of the other MPCitH-based submissions to NIST's call for additional PQC signature schemes. This research may also be of interest to those schemes.

Algorithm 2a and Algorithm 2b describe the SDitH signature generation algorithm, split into *offline* and *online*, respectively, in order to make the distinction in our hardware design description. Dividing the algorithm into two phases (*offline* and *online*) allows us to run these phases in an interleaved fashion (as we describe later in Section 6.3). This way, we completely hide the clock cycles taken by the *offline* phase when we perform multiple signature generation operations.

By profiling signature generation (Algorithm 2a and Algorithm 2b), we note that the Evaluate function inside the ComputePlainBroadcast and PartyComputation can be the main bottleneck in terms of clock cycles. Hence, as described in Section 4.1, we utilise a pipelining approach to optimise the implementation on the underlying arithmetic whilst maintaining minimal area consumption. We observe that the number of times the Evaluate operation is performed in the PartyComputation module (from the *online* phase) is higher than that of the ComputePlainBroadcast module (from the *offline* phase). Hence, the online phase takes more clock cycles, consequently hiding all cycles taken by the *offline* phase.

Algorithm 2a SDitH – Hypercube Variant – Signature Generation (Offline Part)

```
Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$
salt $\leftarrow \{0,1\}^{2\lambda}$, mseed $\leftarrow \{0,1\}^{\lambda}$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$
$\{\mathsf{rseed}[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandSeed}(\mathsf{salt}, \mathsf{mseed}, \tau)$    ▷ See Section 3.2.2
for $e \in [1:\tau]$ do
    $(\mathsf{seed}[e][i])_{i\in[1:2^D]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{rseed}[e])$    ▷ See Section 3.2.9
    acc = 0
    $\mathsf{input\_mshare}[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
    for $i \in [1:2^D]$ do
        if $i \neq 2^D$ then
            $\mathsf{input\_share}[e][i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$
                                                                    ▷ See Section 3.2.1
            acc += $\mathsf{input\_share}[e][i]$
            $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
            for $p \in [1:D]$ : the $p^{\text{th}}$ bit of $i-1$ is zero, do
                $\mathsf{input\_mshare}[e][p]$ += $\mathsf{input\_share}[e][i]$
        else
            acc_wit, acc_beav_ab, acc_beav_c = acc
            $\mathsf{beav\_ab\_plain}[e] = \mathsf{acc\_beav\_ab} + \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], 2dt\eta)$
                                                                    ▷ See Section 3.2.1
            $\mathsf{beav\_c\_plain}[e] = \mathsf{beav\_c\_plain} \leftarrow \mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$
            $\mathsf{aux}[e] = (\mathsf{wit\_plain} - \mathsf{acc\_wit}, \mathsf{beav\_c\_plain}[e] - \mathsf{acc\_beav\_c})$
            $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
        $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$    ▷ See Section 3.2.7
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$    ▷ See Section 3.2.8
$(\mathsf{chal}[e])_{e\in[1:\tau]} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$    ▷ See Section 3.2.3
for $e \in [1:\tau]$ do
    $\mathsf{input\_plain}[e] = (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}[e], \mathsf{beav\_c\_plain}[e])$
    $\mathsf{broad\_plain}[e] \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}[e], \mathsf{chal}[e], (H', y))$    ▷ See Section 4.2
```

Algorithm 2b SDitH – Hypercube Variant – Signature Generation (Online Part)

```
for $e \in [1:\tau]$ do
    for $p \in [1:D]$ do
        $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}[e][p], \mathsf{chal}[e], (H', y), \mathsf{broad\_plain}[e], \mathtt{False})$    ▷ See Section 4.3
$h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$    ▷ See Section 3.2.8
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$
for $e \in [1:\tau]$ do
    $\mathsf{path}[e] \leftarrow \mathrm{GetSeedSiblingPath}(\mathsf{rseed}[e], i^*[e])$    ▷ See Section 3.2.5
    if $i^*[e] = 2^D$ then
        $\mathsf{view}[e] = \mathsf{path}[e]$
    else
        $\mathsf{view}[e] = (\mathsf{path}[e], \mathsf{aux}[e])$
return $\sigma = \left(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\right)$
```

SDitH Signature Verification The signature itself effectively consists of the public elements used in the signature generation: the salt, the transcripts of the τ MPCitH protocols, and a hash of these with a message, m. The job of the verification algorithm, Algorithm 7, is to use these components to re-compute, re-simulate, and re-check, and as such verify the hash values match and thus verify a genuine signature. When we observe the signature verification operation given in Algorithm 7, we note that the modules used in the construction of the signature verification are similar to those of signature generation operation (Algorithm 2a and Algorithm 2b). In this case however we cannot perform the two-phase optimisation, like in signature generation, due to the nature of the algorithm. Consequently, all hardware optimisations we employ are at the level of individual modules (as described in all subsections of Section 3 and Section 4) and optimised scheduling to exploit hardware parallelism (as described in Section 7).

By profiling signature verification (Algorithm 7) we note that the clock cycle bottleneck is Evaluate operation in the PartyComputation function. In Section 4, we detail on how the Evaluate module is implemented in software utilising a large amount of memory versus how we optimise our hardware implementation to present an area-optimised design.

SDitH Parameters We use the same parameters from the SDitH specifications [AFG+23], shown in Table 1. The parameters were derived to target the NIST Security Categories Level 1 (equivalent to the computational hardness of AES-128), Level 3 (similarly AES-192), and Level 5 (similarly AES-256). The SD base fields remain the same for these three, either using GF(2^8) or GF(251) – written GF256 or GF251, respectively – as does the number of secret shares, $N = 256$.

Another parameter that adds to the complexity of the hardware designs is the d splitting, which determines how some of the other parameters are split into smaller sizes. This split happens in L3 and L5 parameters, where $d = 2$, which affects the SampleWitness part of key generation (discussed in Section 3.3), the code length (m), and hamming weight bound (w), which are all divided into two smaller parts. One advantage from a hardware perspective is that we can perform operations on these individual splits in parallel. Similarly, other algorithms can take advantage of these parallel splits, such as the ComputePlainBroadcast and PartyComputation modules discussed in Section 4.2 and Section 4.3.

Table 1: 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

3 SDitH - Hardware Design

Figure 1 illustrates the overall structure of our hardware design and how we have decided to present this in this paper, using a bottom-up approach. This section in particular provides a detailed description of our hardware implementation of base modules. The proceeding sections detail the MPC modules which then all go together to describe our overall key generation, signing, and verifying hardware designs. We note that except for the modular arithmetic modules (described in Section 3.1), all other hardware modules are parametrisable, making them easy to switch between different parameter sets and beyond.

Evaluation Strategy We use the NIST recommended Xilinx Artix 7 xc7a200t-3 as the target FPGA. We verify the functional correctness of our design using the test vectors generated from the reference implementation. We use Time-Area Product (TAP = Time × Configurable Logic Block (CLB) Slices) as an evaluation metric for comparing the performance of our GF256 and GF251 designs, which is used in most of our tables, as is the use of two values side-by-side, such as X/Y, to show GF256/GF251 results.

3.1 Field Arithmetic Operations

As specified in Section 2, SDitH comes with parameters for three security levels, where each security level can be categorised based on the finite fields, namely GF256 and GF251 (i.e. having 8-bit widths).These fields are considered as the base fields of the Syndrome Decoding (SD) instance. Furthermore, these base field SD instances are extended into a larger field (of width 32 bits) which act as a base field for multi-party computation elements, specifically the beaver triplets $([a], [b], [c])$. In this section we will present the hardware designs of these operations.

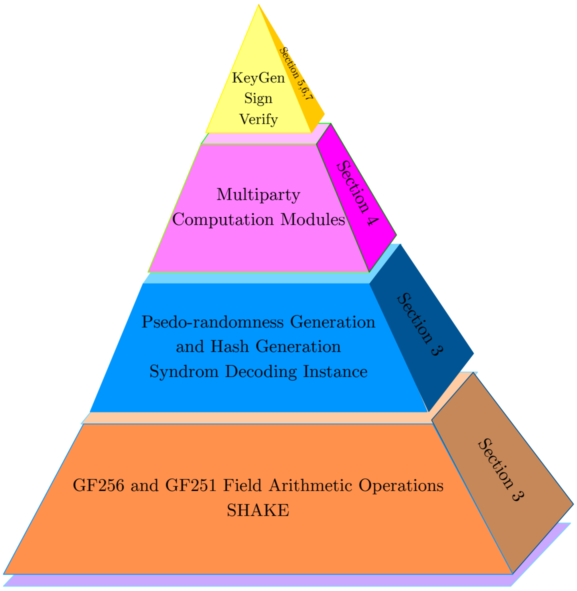

Figure 1: Building blocks and construction of our SDitH hardware design.

3.1.1 $\mathbb{F}_q$ (8-bit) Modular Multiplication and Addition

Modular Multiplication For GF256 multiplication (for the field $F_2[x]/(x^8+x^4+x^3+1)$, we design an LFSR-based optimised multiplication unit inspired by the one described in [SR17]. For the GF251 modular multiplication we use a combination of traditional 8-bit multiplication followed by a modular reduction. We design an optimised modular reduction unit based on Barrett reduction specifically targeting GF251.

Modular Addition It should be noted that for modular multiplication and modular addition, in both underlying arithmetic fields (GF256 and GF251), modular reduction is necessary. In GF256 designs, we do this with a simple with `XOR` operation and LFSRs. For GF251, we use traditional addition and multiplication followed by a modular reduction (Barrett-reduction) since it is a prime field.

3.1.2 $\mathbb{F}_{\text{points}}$ (32-bit) Modular Multiplication and Addition

In addition to the aforementioned GF256 and GF251 fields, the operations involving the multi-party computation elements, such as the beaver triplet operations ($c = a \cdot b$), happen in the extension field $\mathbb{F}_{\text{points}}$ (i.e. $\mathbb{F}_{q^\eta}$) when operating in either GF256 and GF251. The multiplication units in this extension field are constructed using the smaller $\mathbb{F}_q$ multiplication units (described in Section 3.1.1) and involve pipelining. There are two reasons why we pipeline this module (i) to maintain the maximum clock frequency and (ii) to increase the throughput in this component, which is used in multiple modules (e.g. in ComputePlainBroadcast and PartyComputation described in Section 4.2 and Section 4.3, respectively) which involve multiple inputs. Furthermore, for throughput improvements, we decide to pipeline rather than duplicate the hardware modules, because as shown in Table 2, the $\mathbb{F}_{\text{points}}$ arithmetic modules are expensive in terms of their area footprint since our target was an area-optimised design.

For constructing the $\mathbb{F}_{q^\eta}$ (i.e. $\mathbb{F}_{2^{32}}$) multiplications, we first construct $\mathbb{F}_{q^{\eta/2}}$ (i.e. $\mathrm{GF}(2^{16})$) using $\mathbb{F}_q$ multiplications and shown in Figure 2a. We then extend the construction to build to $\mathbb{F}_{q^\eta}$ using the set of $\mathbb{F}_{q^{\eta/2}}$ multiplications. The constructions of $\mathrm{GF}(2^{16})$ and $\mathrm{GF}(2^{32})$ are shown in Figure 2. The additions, in the case of $\mathrm{GF}(256^4)$ are performed using `XOR` operation and in the case of $\mathrm{GF}(251^4)$ are performed using traditional carry based adder followed by Barrett reduction.

3.2 Pseudo Randomness Generation (PRG) and Hash Generation

The SDitH scheme uses a PRG for all its sampling requirements, these being: seed expansion, seed (Merkle) tree expansion, expansion of the parity check matrix, expansion of the MPC challenge, and

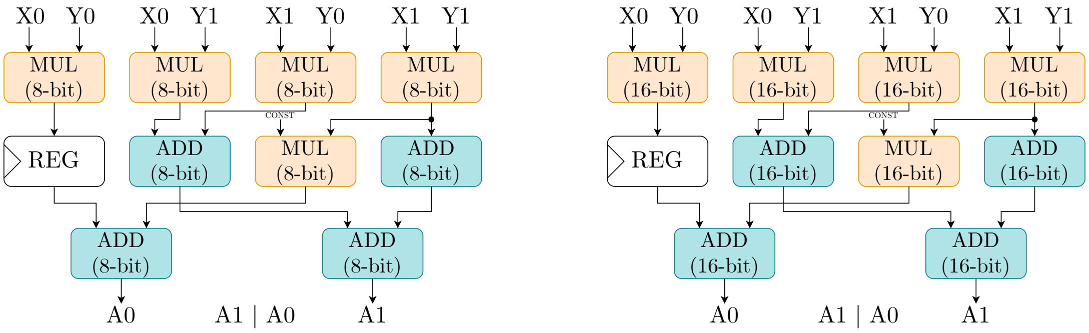

Figure 2: (a) 16-bit multiplications constructed from 8-bit multiplications and additions and (b) 32-bit multiplications constructed from 16-bit multiplications and additions. In both (a) and (b) X0 Y0 A0 represents lower 8 or 16 bits and X1 Y1 A1 represents high 8 or 16 bits.

Table 2: Time and Area results (written as X/Y for GF256/GF251) for modular arithmetic modules targeting the Xilinx Artix 7 `xc7a200t` FPGA.

Operation		**FPGA Utilisation**			**Freq.**	**Latency**	**Time**
	Slices	**LUT**	**FF**	**DSP**	(MHz)	($Kcycles$)	(ms)
		($\times 10^3$)	($\times 10^3$)				
$\mathbb{F}_q$ Mod Add	8/15	8/39	0/39	0/0	–/320	0/3	–/9.37
$\mathbb{F}_q$ Mod Mul	14/42	35/127	26/26	0/0	300/194	2/2	6.67/10.30
$\mathbb{F}_{\text{points}}$ Mod Add	32/161	32/326	0/514	0/11	–/239	0/10	–/41.84
$\mathbb{F}_{\text{points}}$ Mod Mul	280/194	652/437	326/734	0/15	285/237	3/13	10.52/54.00

the expansion of view-opening challenge. It also uses a hash function for commitments and for the Fiat-Shamir transforms. We also adopt the same symmetric-key primitives used in the SDitH specification [AFG+23, Table 3], also see Table 3 which are for NIST security levels L1, L3, and L5, respectively: the SHA3-256, SHA3-384, and SHA3-512 hash functions and SHAKE-128, SHAKE-256, and SHAKE-256 for XOF. We also adopt all SDitH subroutines, most of which can be found in [AFG+23, Section 3.2], however we have also included them in Appendix A.1 for ease of reference.

Table 3: 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

In our work, we use an existing SHAKE-256 module used in [DXN+23, Section 2.1]. The SHAKE-256 module is a parameterised Keccak module that works for different performance parameters (called `parallel_slices`). We use the `parallel_slices = 32` configuration in our work as this yields the best time-area product [DXN+23, Table 1]. However, this SHAKE only performs SHAKE-256, thus we improve and adapt their code (such as modifying the control logic) to make it also work for SHAKE-128.

The SHAKE module[6] is a processor-like design with a 32-bit AXI-Lite interface for input and output. The SHAKE-256 module expects sets of instructions at the initialisation before we load the actual input data for which we want to compute the hash. In SDitH, we use SHAKE in several different modules as described in Section 3.2.1 to Section 3.2.8. For each hash computation, the input and output sizes differ. Irrespective of the different input sizes, the number of clock cycles taken for the Keccak round function inside the SHAKE module always remains constant. This is ensured by keeping a count of the number of blocks (32-bit) loaded at the input and then performing a padding on the rest of the incomplete input blocks. In addition to that, the data inputs to the SHAKE module also come from different modules. All of this combined increases the complexity of the multiplexing logic at the input port of SHAKE, which

[6] From here we use the reference to either SHAKE-128 or SHAKE-256 modules as just simply SHAKE.

affects the maximum clock frequency of the overall design.

Consequently, we design a wrapper around the existing SHAKE module. The wrapper handles all the communications with SHAKE efficiently. It also, interfaces with any BRAM from where it has to pick the data and feed SHAKE for the hash computations. The data-path of the permutation function within the SHAKE module, for the `parallel_slices = 32` mode takes two clock cycle per round. The time and area results for the SHAKE module are shown in Table 4.

The remaining parts of this section detail the different modules where SHAKE is used in SDitH. Although all the following modules use the SHAKE module in their operations, we do not add a separate SHAKE module in each module (which is essential to optimise for area efficiency, since a SHAKE module is expensive in terms of the resource utilisation as depicted in Table 4). Instead we provide an interface for the module to communicate with a common SHAKE module that is shared amongst all the other modules.

3.2.1 SampleFieldElements

Our SampleFieldElements module combines the *Sampling from XOF* and *Sampling field elements* functions specified in Appendix A.1. The SampleFieldElements algorithm takes a seed input and integer number N as input and generates N bytes of output. Depending on the underlying field the sampling process varies. In the GF256 field, the bytes generated from the SHAKE module are directly accepted. Whereas, in the GF251 field, the bytes generated from the SHAKE undergo rejection sampling where a threshold check is performed to see if the generated byte lies in the range $[0, 250]$.

3.2.2 ExpandSeed

The ExpandSeed (described in Appendix A.1) module takes an input of a 2λ-bit salt, λ-bit master seed, and an integer value (N) and generates N λ-bit seeds using the SHAKE module described in Section 3.2. In our hardware design, we use BRAM to store all the generated seeds in blocks of 32-bits. The area results are not accounted for because this module is integrated along with other modules for area optimisation purposes.

3.2.3 ExpandMPCChallenge

The ExpandMPCChallenge (described in Appendix A.1) module is used in both signature generation (Algorithm 2a) and verification (Algorithm 7). The ExpandMPCChallenge module takes as input the h_1 hash, of 2λ bits, and generates τ MPC challenge pairs ($\mathsf{chal} = (r, \varepsilon)$) by expanding h_1 using a PRG. In our hardware implementation module we first feed h_1 into the SHAKE module and generate $2 \times 32 \times \tau$ bits. We then parse the bits and store them in the BRAM. Table 4 shows the area and time performance numbers for our ExpandMPCChallenge module.

3.2.4 ExpandViewChallenge

The ExpandViewChallenge (described in Appendix A.1) module takes the hash output (h_2) of length 2λ bits generated by the Sign Offline (described in Section 6.1) part as the input and generates τ 8-bit integers by expanding h_2 using a PRG. In our hardware implementation module we first feed the h_2 into the SHAKE module and generate $8 \times \tau$ bits. We then parse the bits and store individual locations in the BRAM. Table 4 shows the area and time performance numbers for our ExpandViewChallenge module.

3.2.5 GetSeedSiblingPath

The GetSeedSiblingPath module takes 2λ-bit salt input, λ-bit seed input, and an index i^*, and generates a sibling path of the seed leaf that is indexed by i^* as output. The length of each seed in the seed path is λ bits. The operation is accomplished by feeding the SHAKE module with the salt and seed and then expanding the binary tree similar to that of the Section 3.2.9, but instead of storing all the seeds we only store the sibling seeds in the path until we reach i^*.

3.2.6 GetLeavesFromSiblingPath

The GetLeavesFromSiblingPath module is used in the signature verification algorithm (described in Appendix A.1). It takes 2λ-bit salt, an index i^*, and a seed path as inputs and generates all the leaf shares except the i^* indexed share using the SHAKE module (described in Section 3.2). The generated leaf shares are stored in blocks of 32 bits in a BRAM. This module is implemented as part of the main controller logic in the signature verification module shown in Figure 9 for area optimisation purposes. Consequently, we do not account for its area numbers separately.

3.2.7 Commit

The Commit module takes 2λ-bit salt input, 16-bit execution index, 16-bit share index, and a state input of variable length and generates a hash value of length $2 \times \lambda$ bits using the SHAKE module. In our hardware design, the Commit module is embedded along with the share generation module to avoid latency due to the memory transfers. The Commit module interfaces with the SHAKE module (described in Section 3.2). The control logic for the Commit module collects all the required inputs for the commit computation, arranges it into 32-bit blocks, and stores them in a BRAM. Once all the required inputs are ready, it starts the SHAKE module to compute the hash. The Commit module is embedded along sign_offline and signature verification described in Section 6.1 and Section 7 respectively. Hence, we do not report its performance numbers separately.

3.2.8 Hash_1 and Hash_2

The Hash_1 (h_1) and Hash_2 (h_2) hash computations are called the Fiat-Shamir hashes. Both the hash computation modules in hardware are realised using a BRAM and control logic consisting of a state machine interfaced with the SHAKE module.

The h_1 computation belongs to the offline part of the signature generation and verification operation shown in Algorithm 2a and Algorithm 7, respectively. For the h_1 computation, the control logic gathers part of secret, i.e. seed_H, y, salt, and all the commits, and appends it with byte 0x01 in the most significant part, and computes a hash which generates 2λ-bit output.

The h_2 computation belongs to the online part of the signature generation and verification operation shown in Algorithm 2b and Algorithm 7, respectively. The control logic gathers and appends the message (m), salt, h_1, all outputs from ComputePlainBroadcast (broad_plain), and PartyComputation (broad_share), and appends it with the byte 0x02 in the most significant part and computes a hash and generates a 2λ-bit output.

3.2.9 TreePRG

The TreePRG module takes as input a 2λ-bit salt input and a λ-bit seed and uses the SHAKE module to generate N λ-bit seeds as output. These N seeds correspond to the leaves of binary tree. The nodes of the binary tree are numbered in a hierarchical order where the root seed (i.e. the input seed) is numbered as $i = 1$ and leaf seed on left is numbered with $2i$ and the seed on the right is numbered as $2i + 1$ and the order follows as we grow the tree. A total of 256 seeds are generated. A visual representation of the the TreePRG construction is given in [AFG$^+$23, Figure 2]. We use a BRAM with a width of 32 bits for the storage of the seeds in an incremental order. Table 4 shows the area and time performance numbers for our TreePRG module for L5 parameter set.

All modules presented in Section 3.2 except for the SampleFieldElements modules are constant-time since they have fixed-length inputs and outputs. The SampleFieldElements module behaves in variable time due to the rejection sampling process that is involved. However, this affects only the public information. In addition to this, we note that this variable time behavior of SampleFieldElements is purely because of the nature of the algorithm and is not a shortcoming of the reference software or our hardware implementation. This conforms with the specification and is compliant with the reference software implementation.

Table 4: Pseudo-randomness generation and hash computation area and time results for L5 parameter set targeting the Xilinx Artix 7 `xc7a200t` FPGA.

SDitH	**FPGA Utilisation**				**Freq.**	**Latency**	**Time**
Submodules	**Slices**	**LUT**	**FF**	**BRAM**	(MHz)	($Kcycles$)	(ms)
SHAKE	1,418	4,988	280	0.0	164	74	0.45
SampleFieldElements	138	439	269	0.5	217	350	1.61
ExpandMPCChallenge	30	64	90	1.0	338	610	1.80
ExpandViewChallenge	67	117	107	0.0	305	124	0.41
GetSeedSiblingPath	28	70	52	0.5	262	748	2.85
Hash2	156	495	408	4.0	250	6,734	26.96
TreePRG	44	76	44	8.0	270	27,649	102.44

3.3 SampleWitness

The SampleWitness is one of the essential components of the key generation module in SDitH and is identical for both hypercube and threshold versions of the signature scheme. This module is responsible for the sampling of the d-split SD solution from a witness seed ($\mathsf{seed}_{\mathsf{wit}}$) and building three polynomials namely $\mathbf{Q}$, $\mathbf{S}$, and $\mathbf{P}$ using the modules ComputeQ, ComputeS, and ComputeP, respectively. In our hardware design we achieve this using following process: we first load the $\mathsf{seed}_{\mathsf{wit}}$ into the SampleFieldElements module (described in Section 3.2.1) and generate d fixed-weight polynomials of weight m/d. To build each fixed-weight polynomial we need a position (pos) and a corresponding value (val). Consequently, while we sample from SampleFieldElements module, we sample two sets of random numbers. After generating the fixed weight polynomials, we start ComputeQ and ComputeS modules (described in Section 3.3 and Section 3.3) in parallel. For the $\mathbf{P}$ computation, $\mathbf{Q}$ is needed. The ComputeQ concludes earlier than ComputeS. We then start ComputeP once $\mathbf{Q}$ values become available. Consequently, we hide all the cycles required for ComputeQ and ComputeP by running them in parallel with ComputeS.

In the case of the higher security parameters, for L3 and L5, we use two sets of ComputeQ, ComputeP, ComputeS, and BRAM modules, one for each share, and take advantage of the additional parallelism by running each share operations in parallel. The hardware design for the SampleWitness module is shown in Figure 3. The grey part in the figure is only enabled at compile time for L3 and L5 security levels. The time and area results for the hardware designs of the SampleWitness modules is given in Table 5. We note that the timing required for the L3 parameter set is lower that of L1 because, in L3 and L5 parameter sets, we perform a set of operations in two smaller shares. Thus, we run the two smaller shares in parallel by enabling the grey coloured block in Figure 3 and thus reduce the total time required.

ComputeQ The ComputeQ module takes the w/d sampled non-zero *pos* values as input and maps this list of *pos* values to a polynomial, generating a $w/d+1$ degree polynomial $\boldsymbol{Q}$ output. The process of generating $\boldsymbol{Q}$ involves the multiplication of w/d polynomials of degree one ($\prod_{j=1}^{w/d} X - f_{pos_j}$). From this equation, we note that naïve polynomial multiplications would require a significant amount of storage, several multiplications and additions, and more complex control logic when implemented in hardware. Consequently, rather than using the naïve method, we employ the shift-and-multiply technique. The pseudocode for our shift-and-multiply technique is shown in Algorithm 3. In our hardware design we use the BRAMs in place of the arrays. And we note that all the arithmetic involved is 8-bit modular arithmetic ($\mathbb{F}_q$) described in Section 3.1. Algorithm 3 shows the ComputeQ algorithm.

ComputeS and ComputeP The ComputeS module takes the m/d vector x, of weight w/d, as the input and maps it to values in a polynomial S, of degree $m/d+1$. As specified [AFG+23, Section 3.3], the process of mapping x to form a polynomial is shown in Equation 1. Simply put, $S(x)$ could be obtained by a Lagrange interpolation of the input vector x.

The ComputeP module takes $\boldsymbol{Q}$, from ComputeQ, and $\boldsymbol{S}$, from ComputeS, as inputs and maps them to a polynomial $\boldsymbol{P}$. This mapping could be realised using following equation $P(x) = (Q(x) \cdot S(x)/(\prod_{i=1}^{m/d}(X - x_i))$ as specified in [AFG+23, Section 3.3]. We note that, although the operations such as $(\prod_{i=1}^{m/d}(X - x_i))$ and the computations in ComputeS have several common computations, we still have

Algorithm 3 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]
```

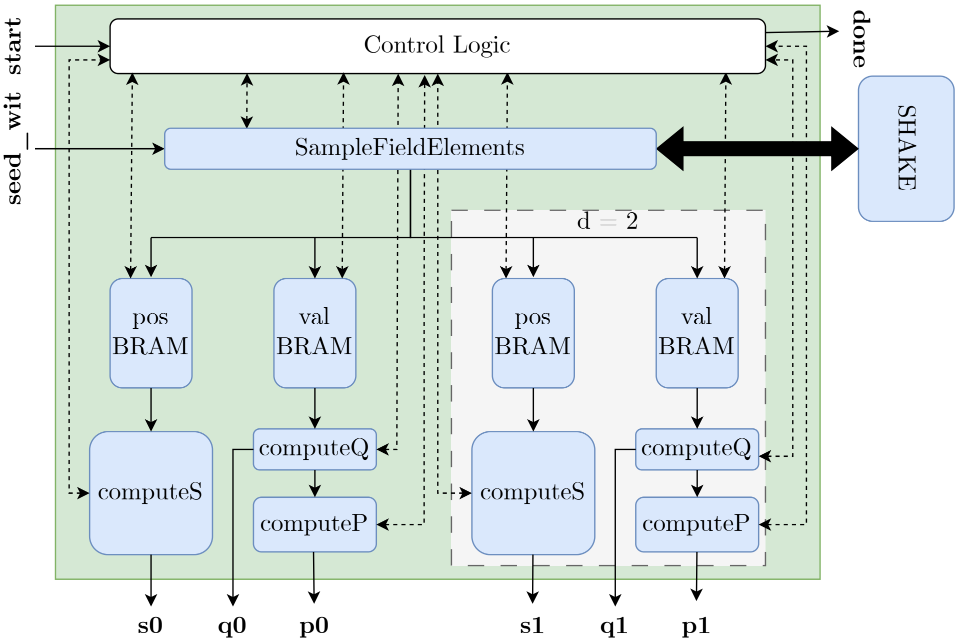

Figure 3: Hardware block design for the SampleWitness Module. The greyed part shows that this part is enabled at compile time, only for security levels L3 and L5, where $d = 2$.

two copies of the underlying multipliers and adders (one for ComputeS and one ComputeQ) to schedule these operation independently at separate times.

In ComputeQ, ComputeS, and ComputeP operations, we use the underlying modular arithmetic described in Section 3.1 which is constant-time. And all iterative operations happen on the fixed compile-time parameters, making the overall modules constant-time.

3.4 Syndrome Decoding (SD) Instance

The H' matrix generation and the matrix vector multiplication module are two essential components of SD instance ($y = H's_a + s_b$). The H' here is a matrix that is sampled from a seed using a PRNG and $s = s_a, s_b$ is a vector. In the SDitH cryptosystem, the dimensions for H' are $(m - k) \times k$ and for s_a, s_b are $k \times 1$, $m - k \times k$ (given in Table 1). Since the sizes of matrix and vector are large, we store them in a BRAM for our hardware design.

From the specification and the reference implementation of SDitH [AFG$^+$23], we note that when the sampling is performed for the H' generation ([AFG$^+$23, Section 3.2.2]) each new sampled element is populated into the matrix by filling it column-wise. Consequently in the hardware, when we generate H', we generate it in a row-major format. Since we have the H' in a row-major format for the matrix-

Table 5: Area and time results (written as X/Y for GF256/GF251) for the SampleWitness 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)	($Kcycles$)	(ms)	
		($\times 10^3$)	($\times 10^3$)						($\times 10^3$)
L1	308/307	917/890	652/668	0/2	4/4	222/198	53/55	0.24/0.28	74/85
L2	319/315	957/913	646/670	0/2	4/4	250/197	32/34	0.13/0.18	41/55
L3	353/336	1,111/974	733/745	0/2	4.5/4.5	229/198	60/62	0.26/0.31	92/105

vector multiplication (for $H' \times s_a$) we use the outer-product method. In this method, each column of the matrix is multiplied with its respective element from the vector and all the resultant vectors are added to get the final output. All the underlying arithmetic (such as the dot products of elements and the element-wise additions) depends on the choice of field. As specified in Section 2, there are two fields, GF256 and GF251. In both fields the operations happen on 8-bit elements. As noted in Section 3.1.1, for GF256, 8-bit modular multiplications use the LFSR method and modular additions use `XOR`. While for the GF251 8-bit multiplications we use one DSP unit per multiplier (if available on the target FPGA) followed by a Barrett reduction, and for the GF251 8-bit adder we use a traditional addition followed by Barrett reduction. For the hardware implementation of SD instance, we consider two approaches namely Sample and Multiply (SaM) On-the-Fly and Sample First and then Multiply (SFTM).

Sample First and then Multiply (SFTM) In the SFTM approach, we first sample elements in a row-major format and store the complete matrix in the BRAM. Since the column sizes are large we break the column into smaller column blocks. The width of each column block is parameterisable. If the choice of the width of the column block makes the last column block partially filled, then we pad the rest of the space in the block with zeros. Once the full matrix is generated, we perform the matrix-vector multiplication ($H' \times s_a$) and vector addition ($H's_a + s_b$). Since we operate on the column blocks, the matrix-vector multiplication operation becomes sequential. The matrix-vector multiplication is also parameterisable, based on the width of each column block of H' matrix. The width of the column block determines how many dot product units and how many adders need to be employed for the underlying arithmetic of matrix vector multiplication operation. We hide the cycles for the vector addition ($H's_a+s_b$) by initializing the BRAM with s_b before we start the matrix-vector multiplication. Figure 4a shows the block design of our SFTM design.

Sample and Multiply On-the-Fly (SaMO) SaMO uses a similar method for the matrix-vector multiplication and the vector addition operations as described in the STFM method. However, in the SaMO method we perform the matrix multiplication as we sample the elements. This method avoids the cost of BRAM storage employed for storing the large H' matrix. In addition to that, we also save several cycles for storing the sampled elements and loading them for the matrix-vector multiplication. To optimise the area utilisation we fix the width column block to the width of the PRNG (SHAKE) module (i.e. 32 bits). Figure 4b shows the block design of our SFTM design.

Table 6 shows the comparison between the two approaches of our SD instance. The choice between STFM and SaMO depends on the availability of all inputs, i.e. the seed for H' and s, while starting the SD operation. In the case of key generation, we can generate H' in parallel to the generation of the vector, s. Consequently, in this case, STFM can be employed where first H' is generated in parallel to s, and then it waits until s is ready to perform the matrix-vector multiplication and vector addition. Whereas in the case of signature generation and verification, the use of SaMO would offer more benefits since we know all the inputs required for the operation beforehand.

As noted in the Section 3.2.1, the process of sampling for GF251 is more stricter than that of the GF256 case. Due to this reason, when we sample for H' generation for the GF256 case, we are able to sample four bytes in a single clock cycle (since SHAKE has a 32-bit interface, described in Section 3.2). Whereas in the case of GF251, we need to ensure that the sampled byte lies in [0,250] to be able to accept it as valid. Due to this sequential nature of sampling, the amount of clock cycles for SD is higher in the case of GF251, as shown in Table 6. We note that due to our SaMO SD approach, we are able to save 90%-99% of the BRAM.

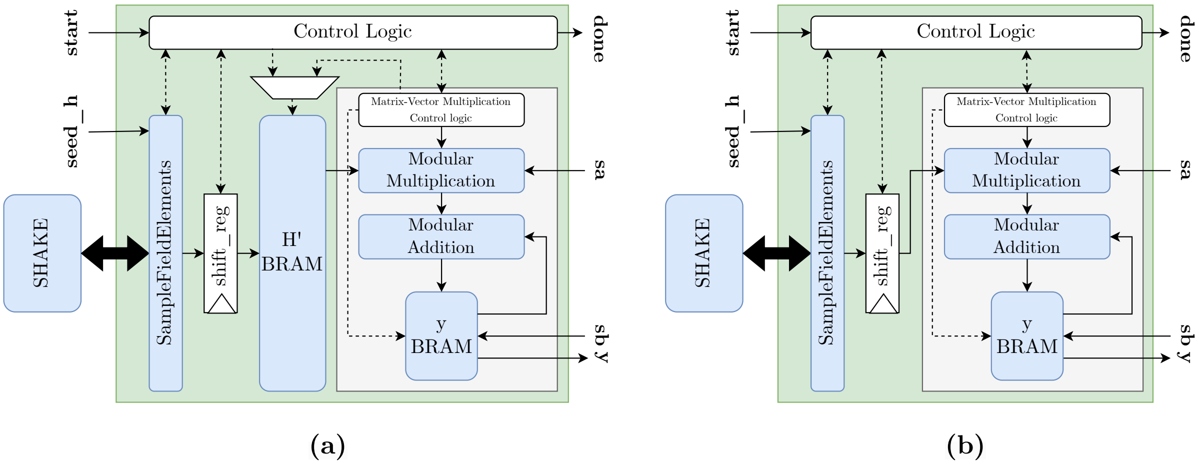

Figure 4: Hardware designs for Syndrome Decoding Instance Module interfaced with the SHAKE module using (a) Sample First and then Multiply (SFTM) and (b) Sample and Multiple On-the-fly (SaMO).

Table 6: Area and time results (written as X/Y for GF256/GF251) for Syndrome Decoding (SD) instance module for all security levels and underlying arithmetic fields targeting the Xilinx Artix 7 `xc7a200t` FPGA.

Param Sets	**Slices**	**LUT** ($\times 10^3$)	**FF** ($\times 10^3$)	**DSP**	**BRAM**	**Freq.** (MHz)	**Latency** ($Kcycles$)	**Time** (ms)	**TAP**
			Sample First and then Multiply (STFM)						
L1	136/116	310/241	279/212	0/4	4.5/4.5	283/267	11/20	0.03/0.07	5/8
L2	151/130	330/295	285/221	0/4	8.5/8.5	286/254	27/49	0.09/0.19	14/25
L3	162/149	348/322	290/227	0/4	16.5/16.5	291/244	48/90	0.16/0.37	27/55
			Sample and Multiply On-the-Fly (SaMO)						
L1	104/103	236/217	188/183	0/4	0.5/0.5	266/261	7/17	0.02/0.06	2/6
L2	114/126	245/266	193/194	0/4	0.5/0.5	260/252	19/41	0.07/0.16	8/20
L3	105/138	241/290	199/200	0/4	0.5/0.5	261/243	35/76	0.13/0.31	14/43

We note that both our SD instances (SFTM and SaMO) are constant-time in the case of GF256. Because the H', S_a, and S_b dimensions are fixed at compile-time, making the matrix-vector multiplication operation constant-time, and the sampled elements for H' from SHAKE are directly accepted without any rejection sampling. Even in the case of the GF251, the matrix-vector multiplication operation is constant-time since the dimensions of H', S_a, and S_b are fixed. However, the sampling for H' matrix generation here involves rejection sampling and thus we observe some variable runtime. This is acceptable and compliant with the reference implementation and specification because H' is a public matrix.

4 Multi-Party Computation (MPC) Modules

This section provides a detailed description of our hardware design of MPC-related modules for SDitH.

4.1 Evaluate

The Evaluate module takes an input polynomial (Q), whose coefficients are in $\mathbb{F}_q$, and a point r, which is in field $\mathbb{F}_{\text{points}}$, and generates an output polynomial evaluation $Q(r)$. The procedure to compute this polynomial evaluation is given in Section A.1 as well as in [AFG+23, Section 3.2.1]. The arithmetic operations involved in Evaluate are a modular exponentiation of r ($r^i \in \mathbb{F}_{\text{points}}$) and an element-wise multiplication and summation ($\sum_{i=1}^{|Q|} Q[i]r^{(i-1)}$). For the hypercube variant of SDitH, $\mathbb{F}_{\text{points}}$ is of width 32 bits and $\mathbb{F}_q$ is of width 8 bits.

Table 7: Area and time results (written as X/Y for GF256/GF251) for Evaluate for all security levels and arithmetic fields targeting the Xilinx Artix 7 `xc7a200t` FPGA. Time results shown for t evaluations, $t = 3$ for L1 and L3, $t = 4$ for L5.

Param	**FPGA Utilisation**					**Freq.**	**Latency**	**Time**	**TAP**
Sets	**Slices**	**LUT**	**FF**	**DSP**	**BRAM**	(MHz)	($Kcycles$)	(μs)	
		($\times 10^3$)	($\times 10^3$)						($\times 10^3$)
L1	580/829	1.3/2.3	1.1/3.0	0/60	0/0	347/200	12/39	35/196	20/175
L3	615/829	1.3/2.3	1.2/3.0	0/60	0/0	330/199	16/30	47/148	17/131
L5	671/1,040	1.5/2.9	1.4/3.8	0/75	0/0	340/201	26/41	78/206	31/230

We note that in the SDitH software[7], the modular exponentiation of all possible outcomes are precomputed and stored in a large lookup table. However, to design and implement such large lookup table on an FPGA is not viable. Thus, in our hardware design we implement the modular exponentiation unit using the pipelined modular addition and modular multiplication modules described in Section 3.1.

For modular exponentiation, we use the square-and-multiply algorithm from [NP17, Algorithm 1], which is the best option in terms of complexity amongst the other alternatives. Although the algorithm shows non-constant time behavior based on the exponent value, in our case the exponents are constants. Therefore, even though individuals may show non-constant time behavior, the overall polynomial evaluation will always remain constant-time. We further note that these operations are on public elements.

The Evaluate module is used as a submodule in ComputePlainBroadcast and PartyComputation modules, described in Section 4.2 and Section 4.3, respectively. In these modules, whenever an Evaluate operation is performed it is performed on t inputs. Accordingly, we take advantage of our pipelined arithmetic units and design an Evaluate unit that can perform polynomial evaluation on t inputs in a pipelined fashion. The time and area utilisation results for our Evaluate hardware design is given in Table 7. From the area results, we note that for L1 and L3, $t = 3$, and hence the area remains almost the same, while for L5, $t = 4$, and consequently the increase in the area can be observed.

4.2 ComputePlainBroadcast

Algorithm 4 describes the process for ComputePlainBroadcast. The module takes the inputs wit_plain polynomials (s_A, $\boldsymbol{Q}$, and $\boldsymbol{P}$), beaver triples ($\boldsymbol{a}, \boldsymbol{b}, c$), challenge ($r$, $\boldsymbol{\varepsilon}$), and the Syndrome Decoding (SD) instance (consisting of the matrix H' and polynomial y) and computes publicly recomputed values of the MPC protocol as well as generating the output broad_plain (consisting of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$).

Our hardware design for ComputePlainBroadcast consists of the process described in Section 3.4 for the SD instance computation (i.e. $s = s_A|y + H's_A$), Evaluate module described in Section 4.1 to compute polynomial evaluation, modular multiplication and addition described in Section 3.1, BRAMs for storing the output broad_plain ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$), and control logic to control the data movement. This module could be used to when the ComputePlainBroadcast is deployed as standalone.

The SDitH sub-modules used in ComputePlainBroadcast could be shared with other operations however (e.g. SD instance module and Evaluate module could be shared in the PartyComputation module). Consequently, we add a parameter to the design which enables an interface and control logic that would allow us to share the sub-modules with the other modules. The reason for sharing the modules is that there is no possibility of parallelism even if we duplicate the modules. Hence, sharing the modules optimises the overall area foot print of our hardware design.

4.3 PartyComputation

Algorithm 5 provides the PartyComputation subroutine. This module takes the inputs wit_share polynomials (s_A, $\boldsymbol{Q}$, and $\boldsymbol{P}$), beaver triples ($\boldsymbol{a}$, $\boldsymbol{b}$, c), challenge (r, $\boldsymbol{\varepsilon}$), SD instance (consisting of the matrix H' and polynomial y), and broad_plain (output from ComputePlainBroadcast consisting of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$) and computes shares broadcast by a party and generates an output broad_shares (consisting of $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and γ).

Similar to the ComputePlainBroadcast module, our PartyComputation hardware module consists of the procedure for computing the SD instance described in Section 3.4, the Evaluate module described

[7] See `https://github.com/sdith/sdith`.

Table 8: Area and time results (written as X/Y for GF256/GF251) for ComputePlainBroadcast module for all security levels and including the underlying arithmetic operations and Evaluate module targeting the Xilinx Artix 7 `xc7a200t` FPGA.

Param	**FPGA Utilisation**					**Freq.**	**Latency**	**Time**	**TAP**
Sets	**Slices**	**LUT**	**FF**	**DSP**	**BRAM**	(MHz)	$(Kcycles)$	(ms)	
		$(\times 10^3)$	$(\times 10^3)$	$(\times 10)$					
L1	723/997	1.7/2.5	1.3/3.2	0/6	0/0	341/191	25/79	0.07/0.41	52/412
L3	1,286/1,890	2.9/4.9	2.5/6.3	0/12	0/0	341/189	19/59	0.05/0.31	70/590
L5	1,383/2,363	3.1/6.1	2.7/7.9	0/15	0/0	340/191	32/83	0.09/0.44	130/1,029

in Section 4.1 to compute polynomial evaluation, modular multiplication, addition, and subtraction described in Section 3.1, BRAMs for storing the output $\mathsf{broad\_share}$ ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, γ), and control logic to control the data movement. Additionally, the module also has a parameter to disable all submodules instantiated inside and enable the sharing of the sub-modules alongside other modules.

Table 9: Area and time results (written as X/Y for GF256/GF251) for PartyComputation 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)	$(Kcycles)$	(ms)	
		$(\times 10^3)$	$(\times 10^3)$	$(\times 10)$					
L1	721/1,407	1.7/3.6	1.6/4.3	0/6	1.5/1.5	280/172	49/157	0.18/0.92	127/1,289
L3	2,273/2,302	3.9/5.9	3.4/7.4	0/12	1.5/1.5	281/171	37/118	0.13/0.69	300/1,591
L5	2,355/2,852	4.0/9.3	3.8/9.3	0/15	2/2	280/174	64/166	0.23/0.96	540/2,728

For further optimisations of the MPC modules, from Algorithm 4 and Algorithm 5, it can be noted that multiple Evaluate functions could be used in parallel to compute the ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$) and ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, γ) values, respectively. This would decrease the overall clock cycle count for each ComputePlainBroadcast and PartyComputation operations but at the cost of increasing the overall resource utilisation (the resource utilisation for Evaluate module is given in Table 7). Since our target was an area-optimised hardware design, we resort to using only one Evaluate module. However, our hardware design of ComputePlainBroadcast and PartyComputation modules could be easily extended to use multiple Evaluate modules.

We note that, as specified in Section 4.1 the Evaluate module is constant-time. And from Algorithm 4 and Algorithm 5 all other operations and iterations happen on fixed compile-time parameters making the both our ComputePlainBroadcast and PartyComputation modules constant-time.

5 SDitH Key Generation

This section provides a detailed description of our hardware implementation for the top-level Key Generation module for SDitH. The Key Generation (KeyGen) module, shown in Algorithm 1, generates a public-key (consisting of seed_H and y) and a secret-key (consisting of $\mathsf{seed}_H, y, \mathsf{wit\_plain}$) from a root seed ($\mathsf{seed}_{\mathsf{root}}$). Our key generation hardware design is shown in Figure 5, where ExpandSeed, SampleWitness, and the SatM Syndrome Decoding (SD) instance (described in Section 3.4) are interfaced with a single SHAKE module (described in Section 3.2) for area optimisation purposes. Hence, it is essential to schedule the usage of the SHAKE module appropriately, and this task is handled by the control logic shown in Figure 5. In our design, first the $\mathsf{seed}_{\mathsf{root}}$ is fed into the ExpandSeed module (described in Section 3.2.2).

The ExpandSeed module generates two seeds $\mathsf{seed}_{\mathsf{wit}}$ and seed_H which are fed into SampleWitness and the SatM SD instance modules, respectively. Then the SHAKE access is assigned to SampleWitness module, the SampleFieldElements inside SampleWitness module uses the SHAKE and $\mathsf{seed}_{\mathsf{wit}}$ and samples the random bits required for generating pos and val required for ComputeQ, ComputeS, and ComputeP (described in Algorithm 6). As soon as the random bits are sampled for generating $\boldsymbol{Q}$, $\boldsymbol{P}$, and $\boldsymbol{S}$, the SHAKE is assigned to the SatM SD instance. After this, the ComputeQ, ComputeP, and ComputeS inside the SampleWitness module and H' matrix generation inside the SD instance are running in par-

Table 10: Area and time results (written as X/Y for GF256/GF251) for KeyGen 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)	($Kcycles$)	(ms)	
		($\times 10^3$)	($\times 10^3$)						
L1	632/799	1.8/2.3	1.1/1.0	0/16	8/9.5	238/196	55/57	0.23/0.29	146/233
L3	770/1,132	2.4/3.3	1.4/1.2	0/22	14.5/14.5	234/194	46/48	0.20/0.24	151/274
L5	856/1,163	2.5/3.4	1.5/1.3	0/22	24.5/25.5	230/197	84/86	0.36/0.44	311/512

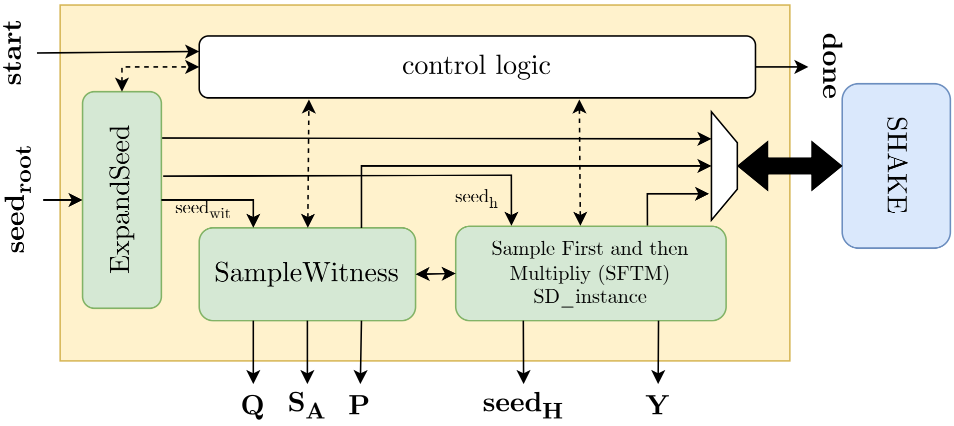

Figure 5: Hardware block design for Key Generation Module interfaced with SHAKE module.

allel. Once both H' matrix and $\boldsymbol{S}$ vector values are ready, then finally matrix-vector multiplication and vector addition module inside the SatM SD instance computes the $H's_A + s_B$ operation. As discussed in Section 3, all underlying modules required in the construction of the KeyGen module except SampleFieldElements elements are constant-time. Overall, our KeyGen module shows only variable time during the initial (public) sampling phase and remains constant for all other operations.

Table 10 shows the area utilisation and timing results for the KeyGen operation for both the GF256 and GF251 fields for all security levels. From the timing results it can be seen that the time taken for L1 security level is higher than L3 that is because, as specified in Table 1, in L3, the d splitting size is two, and enable this to be parallelised.

6 Signature Generation

This section provides a detailed description of our hardware implementation for the top-level signature generation module for SDitH. The signature generation module takes the secret-key consisting of seed_H, y, wit_plain (s_A, $\boldsymbol{Q}$, $\boldsymbol{P}$), and message (m) as inputs and generates signature σ consisting of salt, h_2, view, broad_plain, and com. The algorithm for the signature generation module is given in Algorithm 2a and Algorithm 2b, which we split up signing into two phases, namely, *offline* and *online*. The reason for such division is because we start operating on the secret-key input on Line 24, and message-dependent operations only start on Line 33 during signature generation. All operations before the message is introduced could be performed offline (i.e. precomputed) meaning without the knowledge of the secret-key and message. Dividing the signature generation algorithm into offline and online phases enables the option of interleaving these two phases and consequently hiding all cycles or runtime required by the offline phase in our hardware design.

Furthermore, the division of which operations needs to be in offline phase and online phase also depends on the application where we deploy the signature generation algorithm. For example, in case the application needs to update the secret-key for each new message signed, all operations before Line 23 (in Algorithm 2a) could be in the offline phase and all operations including Line 23 and after have to be considered in the online phase. Whereas, if an application allows signing streams of messages

Table 11: Area and time results (written as X/Y for GF256/GF251) for sign_offline 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	1.1/1.5	3.0/3.9	2.0/3.9	0/6	63/63	240/191	2.4/3.2	9.8/17.2	11/26
L3	1,6/2.4	4.3/6.3	3.1/7.0	0/12	144/144	244/189	3.7/4.7	15.1/25.1	25/61
L5	1.7/2.9	4.6/7.6	3.4/8.6	0/15	180/180	241/191	8.5/10.2	35.36/53.7	63/158

using the same secret-key[8], then operations involving the secret-key processing could also be included in the offline phase hence extending the offline phase until line 32 of signature generation algorithm (in Algorithm 2a) and starting from line 33 until the end can be accounted under the online phase where we process new message signing. And as shown in Algorithm 2a and Algorithm 2b we consider the latter application where the secret-key is not updated often for our hardware optimisation target. We further investigate for the optimal point of division for offline and online by profiling each operation in signature generation. We note that the amount of clock cycles required for the PartyComputation operation in Algorithm 2b is higher than that of the whole sign_offline part and we use this for the optimised SHAKE scheduling process (described in detail in Section 6.3). However, we also note that our hardware design is parameterised to work for all possible cases.

6.1 Signature Generation - Offline Phase

The hardware block design for our offline phase is shown in Figure 6. Our hardware design, assumes that there is a RNG that generates a uniformly distributed random bits and feeds our sign_offline module as inputs salt and mseed. The salt and mseed inputs are fed in to the ExpandSeed module described in Section 3.2.2 and expanded into τ (given in Table 1) seeds (rseed). Each rseed is then extended into a seed tree using the TreePRG module (described in Section 3.2.9). The complete seed tree is stored in a seed_e BRAM. Then the SampleFieldElements module loads each seed from the seed_e BRAM and expands them in to leaf shares. This process is repeated for all seeds except for the last one. Rather than storing the individual shares, the module accumulates all the shares using the modular addition operation. The last seed in the seed_e BRAM expands to the terms of beaver triples (a, b, and c) and are stored in registers beav_a, beav_b, and beav_c.

In addition to that, we also store the input_mshare value that is a serialised input share. An input_mshare value is accumulated only for the cases where the binary representation of the current iteration value have bit positions equal to zero. We accomplish this in our hardware design by designing an add-and-store memory pool (shown in the grey box in Figure 6). It consists of D (the hypercube parameter in Table 1) individual BRAMs where the data is accumulated.

After each leaf share is computed a commitment is generated using the Commit module (described in Section 3.2.7). Once all the commits for all the leaf states of the τ repetitions are generated, a final hash value (h_1) is computed using the Hash_1 module (described in section 3.2.8). The h_1 hash is then expanded into τ challenges (chal which consists of (r, ϵ)) using the ExpandMPCChallenge module described in the Section 3.2.3. Once the chal is ready, the ComputePlainBroadcast (described in Section 4.2) is used to compute the plain values corresponding to the broadcasted shares (broad_plain).

The time and area results for our sign_offline module are shown in Table 11. We note that based on the chosen security level, 27-40% of the time taken for the offline phase is due to the ComputePlainBroadcast module for the arithmetic field GF251. The major contributor to the overall area is the SHAKE module. We note that the BRAM utilisation is high because of the input_mshare storage.

6.2 Signature Generation - Online Phase

In the online phase of signature generation, the broad_plain values from ComputePlainBroadcast are fed into the PartyComputation module (described in Section 4.3) to compute the shares broadcast by a party

[8] Use cases for this include authentication for server-side TLS communications, bank payments, certificate transparency for certificate authorities, and many more.

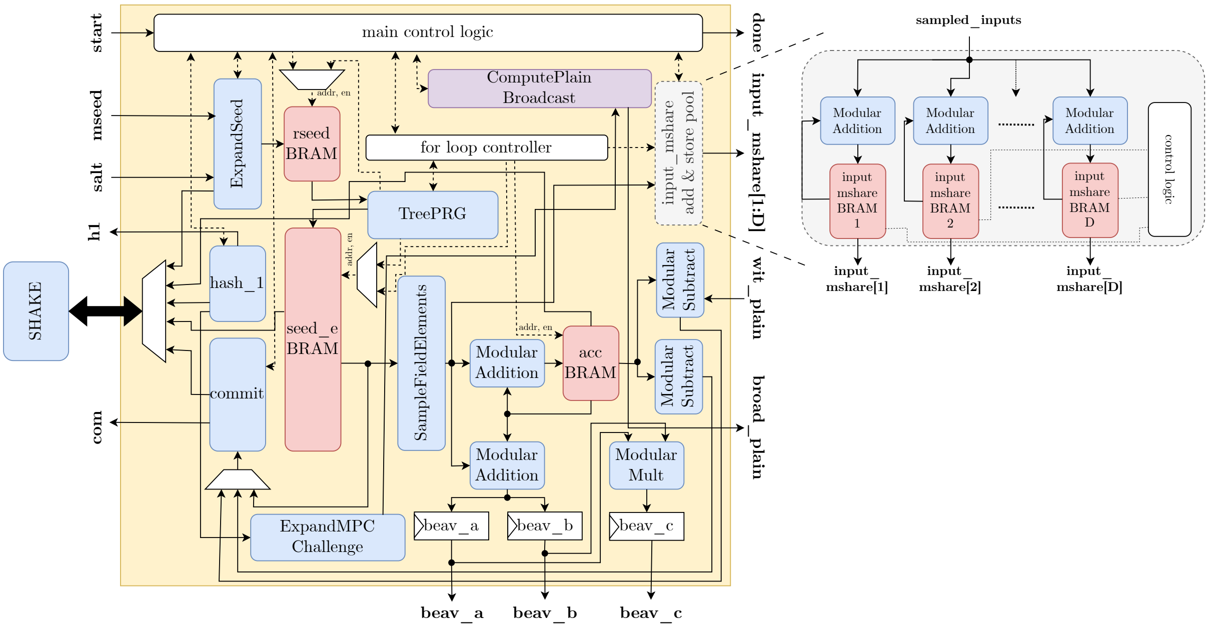

Figure 6: Hardware block design for Offline Phase of Signature Generation Module interfaced with SHAKE256 module.

Table 12: 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	FPGA Utilisation					Freq.	Latency	Time	TAP
Sets	Slices	LUT	FF	DSP	BRAM	(MHz)	$(Mcycles)$	(ms)	
	$(\times10^3)$	$(\times10^3)$	$(\times10^3)$	$(\times10)$					$(\times10^3)$
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

(broad_share). After this, using the input message (m), salt, h_1 generated in sign_offline, broad_plain, and broad_share, we generate a $2 \times \lambda$-bit hash (h_2) using the Hash_2 module described in Section 3.2.8.

Then h_2 is fed into the ExpandViewChallenge module (described in Section 3.2.4) and generates τ 8-bit integers and stores them in a BRAM. These values represent the set of parties to be opened for execution. The for-loop controller shown in Figure 7 then chooses each value from the ExpandViewChallenge module and the root seed (rseed), and generates a sibling path using the GetSeedSiblingPath module described in the Section 3.2.5. The sibling path consists of seeds required at the verifier's end to reconstruct the seed tree for the verification purpose. The sibling path seeds along with the aux values generated in sign_offline are appended together as view. In our hardware design we output these two values separately using two different output ports. The final signature then consists of the $2 \times \lambda$-bit salt, $2 \times \lambda$-bit Fiat-Shamir hash h_2, τ broad_plain polynomials, and τ commits (com).

The time and area results for the sign_online module are shown in Table 12. We note that more than 99% of the clock cycles taken by the sign_online is due to the PartyComputation module in both GF256 and GF251 designs. In the results shown, we do not include the area for SHAKE because in our combined signature design (sign_offline and sign_online), we will be sharing the SHAKE module.

6.3 Interleaved sign_offline and sign_online

As noted in Section 6, Algorithm 2a, and Algorithm 2b, 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

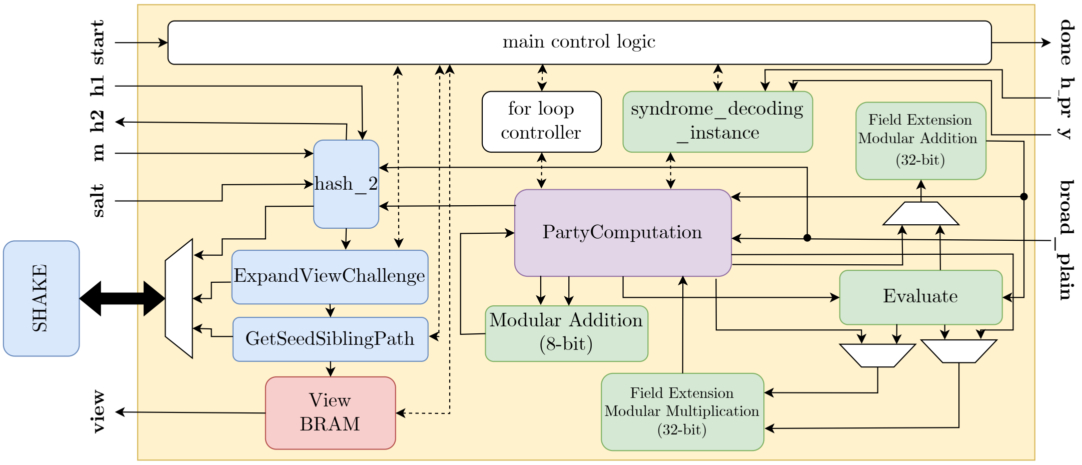

Figure 7: Hardware block design for the online phase of signature generation interfaced with SHAKE module.

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 2a, 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 11 and Table 12) this variable time behavior is completely masked and the overall sign_interleaved module still remains constant-time. From Table 13, 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 9 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 imple-

Table 13: 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

Table 14: 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

mentation. 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 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 2a, 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 9 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 9 to generate the h2_verified output. h2_verified is high if $h_2' == h_2$ if not it stays low. From Table 14, 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.

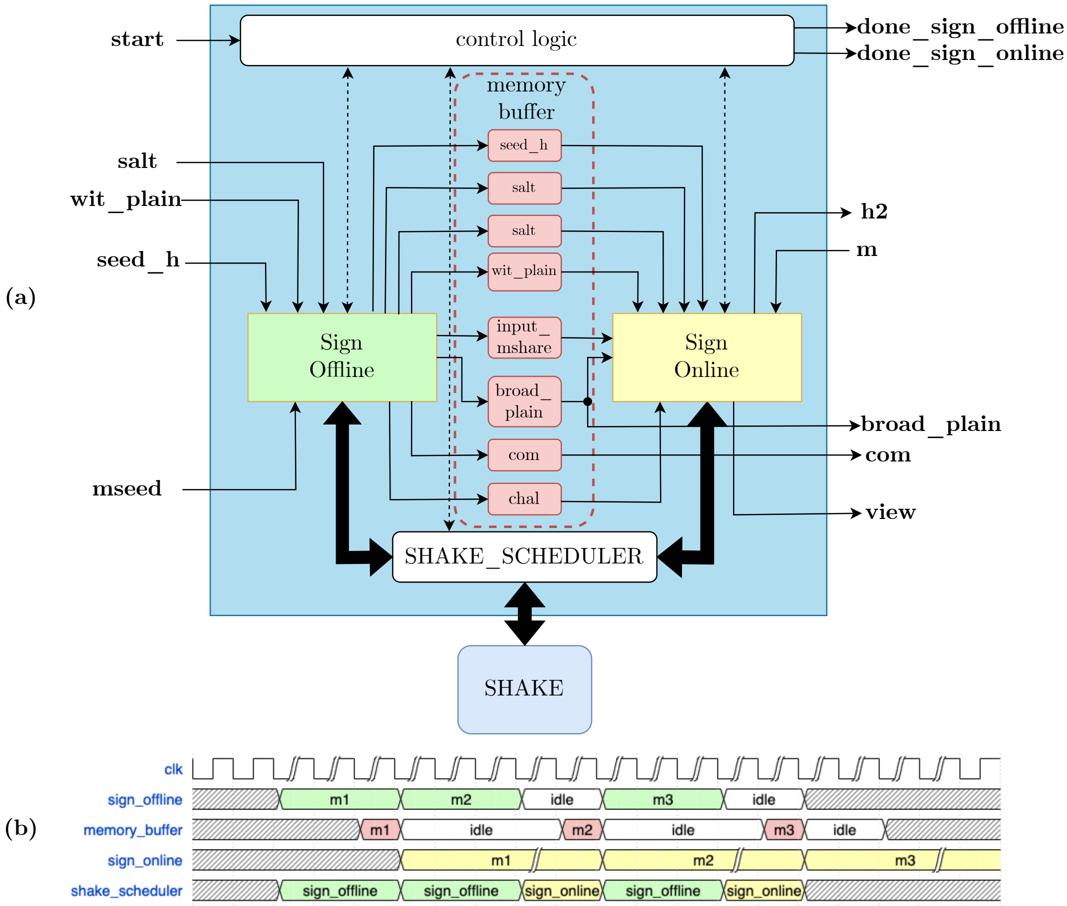

Figure 8: (a) Hardware block design of the full signature generation module where the offline and online phases are working in tandem. (b) The timing diagram showing how the offline and online phases would work in an interleaved fashion while signing multiple messages.

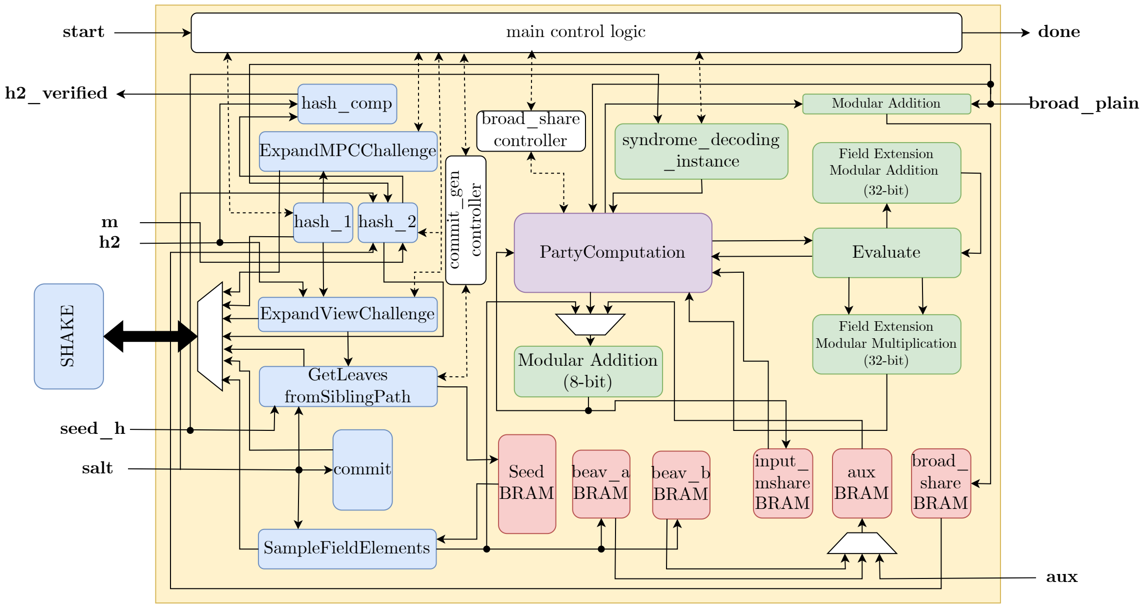

Figure 9: Hardware block design for Signature Verification Module interfaced with SHAKE module.

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 15 and Table 16 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 15, 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 17, 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 implementation, they are able to use Galois-Field-New-Instructions[9] (GFNI) for GF251, which cannot be used in case of GF256.

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

Table 15: Resource 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**
Sets	**LUT**	**FF**	**DSP**	**BRAM**	*(MHz)*
SDitH [Ours]					
SDitH-L1-GF256	16,592	8,778	0	164.5	164
SDitH-L1-GF251	17,423	16,336	196	164.5	164
SDitH-L3-GF256	22,569	13,881	0	356.0	164
SDitH-L3-GF251	29,961	25,794	382	356.0	164
SDitH-L5-GF256	23,323	14,962	0	520.5	164
SDitH-L5-GF251	34,456	31,409	472	521.5	164
PICNIC† [KRR+20]					
PICNIC-L1	90,337	23,105	0	52.5	125
PICNIC-L5	167,530	33,164	0	98.5	125
SPHINCS$^+$-simple‡ [ALC+20]					
SPHINCS$^+$-128s	48,231	72,514	0	11.5	250 & 500
SPHINCS$^+$-128f	47,991	72,505	1	11.5	250 & 500
SPHINCS$^+$-192s	48,725	72,514	0	17.0	250 & 500
SPHINCS$^+$-192f	48,398	73,476	1	17.0	250 & 500
SPHINCS$^+$-256s	51,130	74,576	1	22.5	250 & 500
SPHINCS$^+$-256f	51,009	74,539	1	22.5	250 & 500
Dilithium [ZZW+22]					
Dilithium-L2	29,998	10,336	10	11.0	97
Dilithium-L3	29,998	10,336	10	11.0	97
Dilithium-L5	29,998	10,336	10	11.0	97
LESS [BWM+23]					
LESS-L1 {b,i,s}	54,800	39,900	0	59.5	200
LESS-L3 {b,s}	76,700	57,900	0	102.5	167
LESS-L5 {b,s}	104,300	76,700	0	167.5	143

Table 16: Performance comparison of our SDitH hardware design with other related PQC signature hardware designs for different security levels.

Parameter	**KeyGen**		**Sign**		**Verify**	
Sets	**Latency**	**Time**	**Latency**	**Time**	**Latency**	**Time**
	$(Mcycles)$	(ms)	$(Mcycles)$	(ms)	$(Mcycles)$	(ms)
SDitH [Ours]						
SDitH-L1-GF256	0.055	0.34	6.73	41.04	8.688	52.98
SDitH-L1-GF251	0.057	0.35	21.45	130.77	23.403	142.71
SDitH-L3-GF256	0.046	0.28	7.74	47.17	10.960	66.85
SDitH-L3-GF251	0.047	0.29	24.60	149.99	24.600	149.99
SDitH-L5-GF256	0.083	0.51	17.48	106.60	24.940	152.10
SDitH-L5-GF251	0.086	0.53	45.28	276.07	30.110	183.57
PICNIC [KRR$^+$20]						
PICNIC-L1	–	–	0.03	0.25	0.030	0.24
PICNIC-L5	–	–	0.15	1.24	0.147	1.17
SPHINCS$^+$-simple [ALC$^+$20]						
SPHINCS$^+$-128s	–	–	–	12.40	–	0.07
SPHINCS$^+$-128f	–	–	–	1.01	–	0.16
SPHINCS$^+$-192s	–	–	–	21.40	–	0.10
SPHINCS$^+$-192f	–	–	–	1.17	–	0.19
SPHINCS$^+$-256s	–	–	–	19.30	–	0.14
SPHINCS$^+$-256f	–	–	–	2.52	–	0.21
Dilithium [ZZW$^+$22]						
Dilithium-L2	0.004	0.04	0.03	0.29	0.004	0.05
Dilithium-L3	0.006	0.06	0.04	0.46	0.006	0.06
Dilithium-L5	0.009	0.09	0.05	0.51	0.009	0.09
LESS [BWM$^+$23]						
LESS-L1-b	0.029	0.14	5.20	26.02	5.156	25.78
LESS-L1-i	0.077	0.38	5.13	25.63	5.093	25.47
LESS-L1-s	0.174	0.87	4.17	20.83	4.137	20.69
LESS-L3-b	0.072	0.43	39.24	234.95	39.146	234.87
LESS-L3-s	0.132	0.79	46.22	276.75	46.142	276.85
LESS-L5-b	0.134	0.93	129.89	909.20	129.726	908.08
LESS-L5-s	0.247	1.73	87.16	610.13	87.013	609.09

Table 17: 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

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.

Acknowledgments

This work was supported in part by National Science Foundation grants 2312754 and 2245344.

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, 3, 5, 6, 8, 10–12, 14, 26).

[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, 4).

[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, 23–25).

[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 26).

[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 23–25).

[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 26).

[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 8).

[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 4).

[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 26).

[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, 23–25).

[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 26).

[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 15).

[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 7).

[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, 23–25).

A Appendices

A.1 The SDitH Sub-Routine Procedures

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

- 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.

ComputeS

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

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

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

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 $\text{XOF}.\operatorname{SampleFieldElements}(n)$ samples n pseudorandom elements from $\mathbb{F}_q$. It assumes that the XOF has been previously initialised by a call to $\text{XOF.Init}(\cdot)$. The implementation of the SampleFieldElements routine use the following process. It first generates a stream of bytes $B_1, \dots, B_{n'}$ 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 $B_i \in \{0, 1, \dots, 255\}$. We use the principle of rejection sampling to only select integer values modulo 251, namely we reject byte values in $\{251, \dots, 255\}$. The procedure goes as follows:

```
$i = 1$
while $i \leq n$ do
    $B \leftarrow$ XOF.GetByte()
    if $B \in \{0, 1, \dots, 250\}$ then
        $f_i = B$; $i$ ++
return $(f_1, \dots, f_n)$
```

  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 $\text{ExpandSeed}(\mathsf{salt}, \mathsf{seed}, n)$ initialises the XOF by calling $\text{XOF.Init}(\mathsf{salt} \,\|\, \mathsf{seed})$ and then calls XOF.GetByte() to generate a stream of bytes $B_1, \dots, B_{n\lambda/8}$ which are divided into n output λ-bit seeds $\mathsf{seed}_1, \dots, \mathsf{seed}_n$ as follows:

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

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:

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

where each $\mathsf{chal}\,\mathtt{[}e\mathtt{]}$ 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 $I\,\mathtt{[}1\mathtt{]}, \dots, I\,\mathtt{[}\tau\mathtt{]}$, where $I\,\mathtt{[}e\mathtt{]} \subset [1:N]$ 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 $i^*\,\mathtt{[}1\mathtt{]}, \dots, i^*\,\mathtt{[}\tau\mathtt{]}$ such that $I\,\mathtt{[}e\mathtt{]} = [1:N] \setminus i^*\,\mathtt{[}e\mathtt{]}$ for each execution e. The subroutine then calls

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

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

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

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

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

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.2 The SDitH Subroutine Algorithms

- Algorithm 4 for ComputePlainBroadcast.
- Algorithm 5 for PartyComputation.
- Algorithm 6 for SampleWitness.

Algorithm 4 ComputePlainBroadcast

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

Algorithm 5 PartyComputation

```
Input: input_share := (wit_share, beav_ab_share, beav_c_share), chal, $(H', y)$, broad_plain, with_offset
Output: broad_share
$(s_A, \boldsymbol{Q'}, \boldsymbol{P}) \leftarrow$ Parse(wit_share, $\mathbb{F}_q^k$, $(\mathbb{F}_q^{w/d})^d$, $(\mathbb{F}_q^{w/d})^d$)
$(\boldsymbol{a}, \boldsymbol{b}) \leftarrow$ Parse(beav_ab_share, $(\mathbb{F}_{q^\eta}^d)^t$)
$c \leftarrow$ Parse(beav_c_share, $\mathbb{F}_{q^\eta}^t$)
$(r, \boldsymbol{\varepsilon}) \leftarrow$ Parse(chal, $\mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$)
$(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\beta}}) \leftarrow$ Parse(broad_plain, $(\mathbb{F}_{q^\eta}^d)^t$, $(\mathbb{F}_{q^\eta}^d)^t$)
if with_offset is True then
    $s = (s_A \mid y + H's_A)$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, 1)$
else
    $s = (s_A \mid H's_A)$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, 0)$
$\boldsymbol{S} \leftarrow$ Parse($s$, $(\mathbb{F}_q^{m/d})^d$)
for $j \in [1:t]$ do
    $v[j] = -c[j]$
    for $\nu \in [1:d]$ do
        $\boldsymbol{\alpha}[j][\nu] = \boldsymbol{\varepsilon}[j][\nu] \otimes$ Evaluate($\boldsymbol{Q}[\nu], r[j]$) $+ \boldsymbol{a}[j][\nu]$      ▷ See Section 4.1
        $\boldsymbol{\beta}[j][\nu] =$ Evaluate($\boldsymbol{S}[\nu], r[j]$) $+ \boldsymbol{b}[j][\nu]$      ▷ See Section 4.1
        $v[j]$ += $\boldsymbol{\varepsilon}[j][\nu] \otimes$ Evaluate($F, r[j]$) $\otimes$ Evaluate($\boldsymbol{P}[\nu], r[j]$)      ▷ See Section 4.1
        $v[j]$ += $\bar{\boldsymbol{\alpha}}[j][\nu] \otimes \boldsymbol{b}[j][\nu] + \bar{\boldsymbol{\beta}}[j][\nu] \otimes \boldsymbol{a}[j][\nu]$
        if with_offset is True then
            $v[j]$ += $-\boldsymbol{\alpha}[j][\nu] \otimes \boldsymbol{\beta}[j][\nu]$
broad_share = Serialize($\boldsymbol{\alpha}, \boldsymbol{\beta}, v$)
return broad_share
```

Algorithm 6 SampleWitness

```
Input: $\mathsf{seed}_{\mathsf{wit}} \in \{0,1\}^\lambda$
Output: $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$
$(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P}) \leftarrow$ Init($(\mathbb{F}_q^{w/d+1})^d$, $(\mathbb{F}_q^{m/d})^d$, $(\mathbb{F}_q^{w/d})^d$)
XOF.Init($\mathsf{seed}_{\mathsf{wit}}$)
for $\nu \in [1:d]$ do
    pos$[\nu] \leftarrow$ XOF.Sample($\{J \subseteq [1:m/d]\ ;\ |J| = w/d\}$)
    val$[\nu] \leftarrow$ XOF.Sample($(\mathbb{F}_q^*)^{w/d}$)
    for $i \in [1:m/d]$ do
        $\boldsymbol{x}[\nu][i] = \sum_{j \in [1:w/d]}$ val$[\nu][j] \cdot$ (pos$[\nu][j]$==$i$)

    $\boldsymbol{Q}[\nu] =$ ComputeQ(pos$[\nu]$)      ▷ See Section 3.3
    $\boldsymbol{S}[\nu] =$ ComputeS($\boldsymbol{x}[\nu]$)      ▷ See Section 3.3
    $\boldsymbol{P}[\nu] =$ ComputeP($\boldsymbol{Q}[\nu], \boldsymbol{S}[\nu]$)      ▷ See Section 3.3
return $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$
```

A.3 The Signature Verification Algorithm

Due to the similarities between the Signature Generation algorithm (Algorithm 2a and Algorithm 2b) and the Signature Verification algorithm (Algorithm 7) we have omitted the pseudo-code from the main body of the paper and included it instead in the appendix below.

Algorithm 7 SDitH – Hypercube Variant – Verification Algorithm

```
Input: a public key $pk = (\mathsf{seed}_H, y)$, a signature $\sigma$ and a message $m \in \{0,1\}^*$
Parse $\sigma$ as $\Big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\Big)$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$    ▷ See Section 3.2.4
for $e \in [1:\tau]$ do
    $(\mathsf{seed}[e][i])_{i\in[1:2^D\backslash i^*[e]]} \leftarrow \mathrm{GetLeavesFromSiblingPath}(i^*[e], \mathsf{salt}, \mathsf{path}[e])$    ▷ See Section 3.2.6
    for $i \in \{2^D \backslash i^*[e]\}$ do
        if $i \neq 2^D$ then
            $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
        else
            $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
        $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$    ▷ See Section 3.2.7
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$    ▷ See Section 3.2.8
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$    ▷ See Section 3.2.3
for $e \in [1:\tau]$ do
    $\mathsf{input\_mshare}^*[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
    for $i \in [1:2^D \backslash i^*[e]]$ do
        if $i \neq 2^D$ then
            $\mathsf{input\_share}[e][i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$
                ▷ See Section 3.2.1
        else
            $\mathsf{beav\_ab\_plain}[e][2^D] = \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][2^D], 2dt\eta)$
                ▷ See Section 3.2.1
            $\mathsf{input\_share}[e][2^D] = (\mathsf{aux}[e] \mid \mathsf{beav\_ab\_plain}[e][2^D])$
        for $p \in [1:D]$ : the $p^{\text{th}}$ bit of $i-1$ and $i^*[e]$ are different do
            $\mathsf{input\_mshare}'[e][p]$ += $\mathsf{input\_share}[e][i]$
    for $p \in [1:D]$ do
        if the $p^{\text{th}}$ bit of $i^*[e]$ is 1 then
            $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal},$    ▷ See Section 4.3
                $(H', y), \mathsf{broad\_plain}, \mathtt{False})$
        else
            $\mathsf{broad\_share}[e][p] = \mathsf{broad\_plain}[e] - \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal},$
                $(H', y), \mathsf{broad\_plain}, \mathtt{True})$
                ▷ See Section 4.3
$h_2' = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.    ▷ See Section 3.2.8
return $h_2 \overset{?}{=} h_2'$
```