A Billion Hard CRYSTALS

Automated Design Space Exploration of Mixed-Domain Masking for PQC in Hardware

Fabian Buschkowski[1], Niklas Höher[1], Pascal Sasdrich[1] and Tim Güneysu[1,2]

[1] Ruhr University Bochum, Bochum, Germany, firstname.lastname@rub.de
[2] DFKI GmbH, Bremen, Germany

Abstract. Lattice-based post-quantum cryptographic schemes such as ML-KEM or ML-DSA combine bitwise operations, which are most efficiently masked using Boolean secret sharing, with arithmetic operations modulo q, which are cheaper to mask using an additive sharing in the respective field or ring. Previous work has automated the generation of Boolean-masked hardware circuits as well as the design space exploration process, but to the best of our knowledge, no similar generic framework exists for hardware designs requiring mixed masking domains.
Such designs pose three additional challenges. First, share-wise arithmetic requires no registers to be secure, leading to long critical paths without deliberate pipelining. Second, the conversions required at domain boundaries increase both complexity and susceptibility to human error. Finally, generation of masking randomness from a traditional PRNG requires post-processing, e.g., via rejection sampling, to achieve uniformity modulo q, which results in a variable output rate.
We solve these issues and integrate our scheme-agnostic solutions into the masking and design space exploration framework HADES. An efficient pre-synthesis critical path estimation, which is around 60 times faster than synthesis with traditional timing analysis, enables effective exploration of mixed-domain masked circuits without compromising the maximum clock frequency. Our adapted gadget-based masking procedure tracks masking domains and inserts pipelined conversions at domain boundaries automatically, or lets the template developer place resource-shared conversions manually. Lastly, a Markov-chain model of probabilistic masking randomness generation and buffering yields cost-minimal design configurations that guarantee stall-free masked operation except for a user-defined failure probability.
We evaluate our methodology for multiple relevant moduli and implement an arbitrary-order masked ML-KEM decapsulation core as a case study for our open-source framework. Its area-optimized configurations represent, to the best of our knowledge, the most compact reported ML-KEM designs, with 54.1 kGE unmasked and 125.4 kGE at first order using the NanGate45 library. Jointly optimizing for area and critical path yields competitive clock frequencies of 325–429 MHz for at most 31 % more area, which is still smaller than all previous designs. Practical side-channel security at both first and second masking order is validated through FPGA-based TVLA measurements of the full decapsulation routine using 10 million traces each.

Keywords: Design Space Exploration · Hardware Implementations · ML-KEM · Higher-Order Masking · Arithmetic Masking · Masking Conversions · TVLA

1 Introduction

Driven by the threat of large-scale quantum computers, the design and efficient implementation of Post-Quantum Cryptography (PQC) has received significant research attention

over recent years. Due to the inherent complexity of these schemes, implementing PQC in hardware requires developers to evaluate a plethora of design choices that often counteract each other, making manual optimization tedious and error-prone. Security against physical implementation attacks such as power side-channel analysis [KJJ99] adds another dimension to the design space. One of the most commonly applied and extensively studied methods to protect cryptographic implementations against such attacks is *masking* [CJRR99], which requires careful transformation of the unprotected circuit. Manually applying this technique is a complex task that is highly susceptible to human error. Therefore, multiple tools such as AGEMA [KMMS22] or EASIMask [BSG23] have been proposed to automate this process, while tools such as HADES [BLH+25] and the recently published TETRIS [SPBK26] include additional Design Space Exploration (DSE) capabilities.

One major drawback of these existing tools, however, is the focus on Boolean masking, which is efficient for common symmetric schemes, but rather inefficient for asymmetric schemes, including lattice-based PQC algorithms that rely on complex arithmetic operations such as polynomial multiplications using the Number-Theoretic Transform (NTT). For example, the proof-of-concept masked Kyber design presented by Buschkowski et *al.* as part of the original HADES publication [BLH+25] has a latency overhead of almost 100 % when comparing a first-order masked implementation with an unmasked implementation. Arithmetic masking modulo q can provide significant speed-ups in such cases since linear operations with regard to the sharing, i.e., additions of two secret values or multiplications of secret values by a public value, can be implemented share-wise, instead of having to rely on Boolean-masked arithmetic modules. Some operations, however, remain cheapest to mask using a Boolean sharing, including hashing or sampling within ML-KEM, which inherently leads to designs with mixed masking domains.

However, to this day, tool support for automatically applying arithmetic masking to hardware circuits is very limited. The recently presented HOPE ML-KEM [CNC26] supports generation of side-channel protected designs, partially using arithmetic masking, but it is specifically tailored to ML-KEM. Furthermore, it supports only first-order and third-order masking and leaves sensitive parts of the decapsulation routine unmasked, which raises security concerns, as discussed in Section 7.2.1. As a result, the need for security-aware tooling for the efficient and side-channel protected implementation of arbitrary PQC schemes in hardware still persists. Such a mixed-domain masking tool needs to solve three main problems that tools which only operate on a Boolean sharing do not face:

First, gadget-based masking requires a register stage in every non-linear gadget. Therefore, the critical path of resulting circuits is inherently limited by the logic depth of strictly linear layers, e.g., MixColumns within an AES core. As these are usually quite shallow for Boolean-masked designs, maximum clock frequency is commonly not a concern. In the original HADES publication [BLH+25], all first-order masked symmetric primitives run at over 430 MHz without any explicit critical-path optimizations. This does not transfer to arithmetic masking since linear operations are now not limited to simple XOR and NOT gadgets, but can include arbitrary modular addition, subtraction, or multiplication by a public value. As a result, the clock frequency quickly drops without optional pipeline stages being inserted, adding another dimension to the design space.

Second, a conversion between Boolean and arithmetic sharings of secret values is required when masking domains are crossed. While conversion algorithms themselves are well studied [CGV14, BC22, SH24, NDKV25], their integration into hardware circuits still requires careful consideration of pipelining, protection against glitches, as well as potential interactions with other parts of the circuit, particularly if a single conversion module is utilized at multiple positions within an algorithm.

Third, B2A conversions as well as arithmetically masked field multiplications require fresh uniform elements from $\mathbb{Z}_q$ in every clock cycle, which rejection sampling produces at a variable rate for non-power-of-two q. Buffering valid elements can compensate this

and therefore simulate a fixed-rate randomness source. However, to our knowledge, there has been no joint formal analysis of sampling configurations and required buffer sizes to guarantee that a certain number of random elements is available in every clock cycle at minimal cost, except for a user-defined failure probability.

Our Contribution. In this work, we solve these three problems independently of the implemented target scheme and integrate our solutions into the HADES framework, as highlighted in Figure 1. A template developer writes masking-order-agnostic SpinalHDL templates for a cryptographic scheme once and annotates masking domains within the design hierarchy; a user then selects the targeted masking order and the optimization goal; HADES automatically explores the design space using pre-synthesis performance estimation, masks the best identified configurations, inserts required masking conversions and a wrapper for efficient handling of rejection sampling, and finally emits Verilog or VHDL code. To the best of our knowledge, this makes HADES the first tool that automatically generates arbitrary-order hardware circuits with mixed Boolean and arithmetic masking. In detail, our contributions are as follows:

- **Pre-synthesis critical path estimation** (Section 3.1): Our two stage estimation process outperforms classical post-synthesis timing analysis flows by up to factor 60 in runtime while still providing results which are accurate enough for effective DSE.
- **Automated mixed-domain masking** (Sections 3.2–3.4): As part of our modified HADES framework, we provide a library of arithmetic gadgets over $\mathbb{Z}_q$, which can be used for automated gadget-based masking. HADES furthermore automatically analyzes the masking domains of connected templates and by default inserts a pipelined masking conversion at every domain boundary.
- **Randomness handling over** $\mathbb{Z}_q$ (Section 4): We provide a methodology to reason about the exhaustion probability of randomness buffers depending jointly on the sampling configuration and the buffer size using a Markov-chain model. This enables us to determine cost-minimal randomness generation configurations.
- **ML-KEM case study** (Sections 5 and 7): We provide an ML-KEM decapsulation template combining Boolean and arithmetic masking techniques, resulting, to the best of our knowledge, in the most compact published design yet, both in the unmasked and in the masked variants, when optimizing for area during the DSE process.
- **Detailed full-decapsulation leakage assessment** (Section 7.1): While most previous side-channel protected designs of PQC KEMs resorted to evaluation of individual components or used at most 200 000 traces for evaluation of the complete decapsulation routine [BAA23, DAR+26, BDK+21, AMD+21], we include practical TVLA measurements using 10 million traces for both a first-order masked and, to the best of our knowledge for the first time, a second-order masked ML-KEM design, showing no signs of unexpected side-channel leakage.
- **Open-source release**: The extended HADES framework as well as the ML-KEM templates are available publicly at `[URL REDACTED]`.

The first three contributions do not depend on the ML-KEM case study and are evaluated separately in Section 6. We want to emphasize that the resulting framework can be re-used for arbitrary schemes, e.g., HQC or ML-DSA, in the future.

2 Preliminaries

In this section, we provide the required background on Boolean and arithmetic masking in hardware, the existing HADES framework, as well as a short overview of the NIST-standardized ML-KEM algorithm, focusing on the decapsulation procedure.

2.1 Masking

Masking [CJRR99] is one of the most popular countermeasures against side-channel attacks due to its provable security properties. Based on secret sharing [Sha79], the core idea of this approach is to split up all secret values into multiple statistically independent *shares* so that only the combination of all shares yields any information about the original value. Computations are only performed on this shared representation in order to break the correlation between the processed secret data and any physical properties of a device during execution, e.g., its power consumption. Security is commonly argued in the d-probing model [ISW03] which allows an adversary to observe up to d intermediate values. If the adversary gains any information about the distribution of the secret, the design is considered insecure. This has subsequently been extended to account for physical effects like glitches and transitions, leading to the robust d-probing model [FGP+18].

To achieve d-th order security in practice, a secret value x is split into $d+1$ shares $x_0, ..., x_d$ such that $x_0 \circ x_1 \circ ... \circ x_d = x$ and any d shares are independent of x. For the frequently used Boolean masking, $\circ$ is instantiated with the XOR operation and the sharing of a value is performed by choosing d uniformly random shares $x_0, ..., x_{d-1}$ while setting $x_d = x \oplus \left(\bigoplus_{i=0}^{d-1} x_i\right)$. Even though Boolean masking is an efficient choice to implement Boolean functions such as many cryptographic S-Boxes, it has a significant performance overhead for more complex arithmetic functions such as field multiplications or the NTT. For these applications, arithmetic masking is a more suitable option, which instantiates $\circ$ with addition in $\mathbb{Z}_q$ where q is a prime or a power of two, depending on the use case. To share a secret arithmetically, d random shares are sampled uniformly from $\mathbb{Z}_q$ while the last share is computed as $x_d = x - \sum_{i=0}^{d-1} x_i \bmod q$.

Since many Boolean functions cannot be efficiently implemented using arithmetic masking, it is often necessary to convert shares from Boolean to additive domain and vice versa. While Goubin [Gou01] and Coron et *al.* [CGV14] presented the initial first-order and arbitrary-order masking conversions respectively, multiple later works [CGTV15, SPOG19, BC22] have improved the original results by generalizing the approach to arbitrary moduli q and reducing the overall conversion complexity for specific cases. Dedicated hardware conversion modules have been proposed for first- and second-order only by Shahmirzadi and Hutter [SH24] as well as for arbitrary masking orders by Norga et *al.* [NDKV25]. Due to the generally high overhead of masking conversions, it is advisable to analyze whether the performance improvements due to cheaper operations in arithmetic domain outweigh the cost of the conversion itself.

As transforming an unprotected hardware design into a masked circuit is a complex and error-prone task, the concept of *gadget-based* masking was developed. This technique relies on securely masked reusable atomic building blocks, referred to as *gadgets*, which can be used to construct larger circuits based on their composition. In the case of Boolean masking, these gadgets commonly correspond to simple AND, XOR, and NOT gates. Notable examples include the HPC1 and HPC2 family of gadgets introduced by Cassiers et *al.* [CGLS21] and the low-latency HPC3 family of gadgets by Knichel and Moradi [KM22]. For most common gadget families, linear operations, i.e., field addition and negation, can be performed trivially on individual shares, while non-linear operations, i.e., field multiplication, require additional uniformly distributed field elements to remain secure due to share recombination. Theoretical security guarantees for resulting designs are

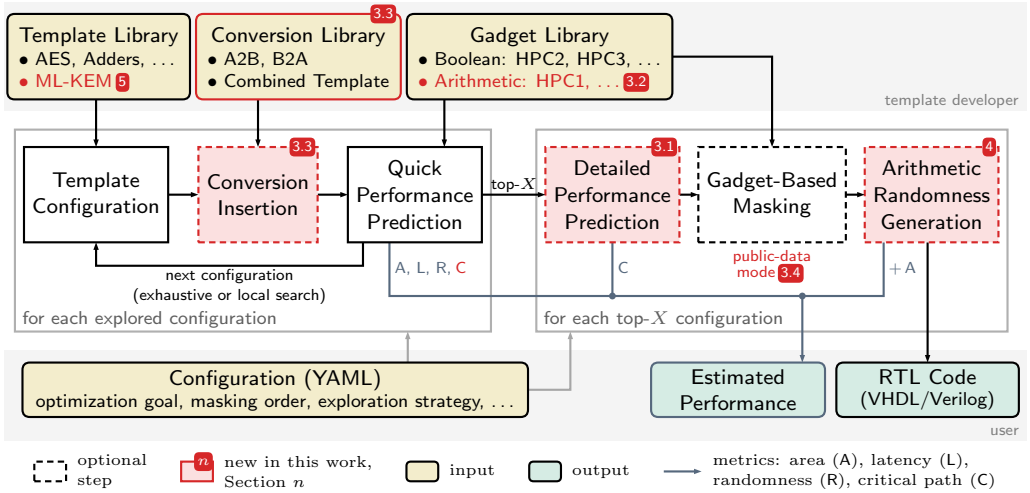


Figure 1. Overview of the complete HADES DSE and masking flow. All changes and extensions made in this work are highlighted in red. Optional steps are marked with dashed lines.

based on corresponding composability notions, most importantly Probe-Isolating Non-Interference (PINI) [CS20, CGLS21], which define a set of rules that need to be followed during gadget composition. Some conversions such as the ones presented in [BC22] are proven to fulfill the PINI notion, which theoretically allow for secure integration into designs consisting of Boolean and arithmetic PINI gadgets.

2.2 HADES

The HADES framework [BLH+25] provides an automated tool for the efficient and security-aware DSE of cryptographic primitives in hardware. Its general design and exploration flow is shown in Figure 1, with all extensions presented as part of this work being highlighted in red. At its core lies an extendable library of configurable hardware *templates* written in the hardware construction language SpinalHDL, which correspond to reusable building blocks similar to Verilog modules with parameters or VHDL entities with generics. A template can for example correspond to a generic adder or an AES S-box. Multiple templates are then composed to form a larger template hierarchy representing full cryptographic primitives. Since templates are masking-order agnostic, they need to be developed only once by a *template developer* who also provides all metadata required by the framework, such as a cycle-accurate latency formula.

Inside of a template, HADES differentiates between a set of immutable, user-defined parameters, that define the black box functionality, e.g., the bit width of an adder, and mutable design aspects which only change the internal implementation of a template, e.g., the architecture of the adder, in a way which is transparent to the user. The set of all design aspects within the template hierarchy forms the overall design space which the framework explores.

Using this supplied template library, a *user* is afterward able to automatically generate a hardware implementation of any supported algorithm by providing an appropriate YAML configuration file which defines, beyond other aspects, the targeted optimization goals and the optional side-channel protection order. Arbitrary-order side-channel security is achieved through means of automated gadget-based masking. The HADES framework then outputs synthesizable Verilog or VHDL code of the best-performing configurations alongside their estimated performance metrics. The list of supported optimization targets currently consists of area (A), latency (L), randomness usage per cycle (R), as well as

Algorithm 1: Kyber decapsulation

Input: $c = (\overline{\mathbf{u}}, \overline{v}) \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}, sk = (\hat{\mathbf{s}}, \hat{\mathbf{t}}, \rho, h, z) \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$

begin
- $\mathbf{u} := \mathsf{Decompress}_q(\overline{\mathbf{u}}, d_u)$
- $v := \mathsf{Decompress}_q(\overline{v}, d_v)$
- $m' := \mathsf{Compress}_q(v - \mathsf{INTT}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1)$ `/* end of decryption */`
- $(\bar{K}', r') := \mathsf{SHA3\text{-}512}(m' || h)$
- expand $\hat{\mathbf{A}}^T$ from ρ using SHAKE-128
- use r' to sample $\mathbf{r}, \mathbf{e_1}, e_2$ from a centered binomial distribution
- $\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
- $\mathbf{u}' := \mathsf{INTT}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e_1}$
- $v' := \mathsf{INTT}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(m', 1)$
- $(c_1, c_2) := (\mathsf{Compress}_q(\mathbf{u}', d_u), \mathsf{Compress}_q(v', d_v))$ `/* end of encryption */`
- **if** $(c_1, c_2) = (\overline{\mathbf{u}}, \overline{v})$ **then return** $K := \mathsf{SHAKE\text{-}256}(\bar{K}' || \mathsf{H}(c))$
- **else return** $K := \mathsf{SHAKE\text{-}256}(z || \mathsf{H}(c))$

end

product metrics denoted as a combination of these acronyms with the suffix P which combine multiple targets, e.g., in the form of the Area-Latency Product (ALP).

For each analyzed configuration, HADES instantiates the template hierarchy and efficiently estimates these performance metrics without performing traditional logic synthesis. Small design spaces can be explored exhaustively while a probabilistic local search algorithm is utilized for larger templates. More precisely, a set of random configurations is drawn and exhaustively evaluated. Individual design aspects of the best-performing initial sample are then transformed one at a time through a sequence of locally optimal changes until no further improvements with regard to the optimization target are possible. The evaluation in [BLH+25] showed that such a local optimum is close to the global optimum in most cases.

While a wide range of cryptographic algorithms is already supported including many symmetric primitives like AES, Keccak, and ChaCha20, the original version of the HADES framework supports only Boolean masking using the HPC2 and HPC3 gadget families. A proof-of-concept implementation of the round-3 Kyber specification was additionally presented in [BLH+25], which demonstrated the feasibility of fully masking PQC designs using Boolean gadgets while simultaneously showcasing the resulting performance penalties.

2.3 ML-KEM

ML-KEM has been standardized by National Institute of Standards and Technology (NIST) [Nat24] based on the Kyber submission [ABD+21]. Its security is based on the application of the Fujisaki-Okamoto (FO) transform to the CPA-secure Kyber public key encryption scheme, so its decapsulation consists of a decryption, a deterministic re-encryption, and a check for equality of the two ciphertexts. Calculations in ML-KEM are done over the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 3329$. The three different standardized parameter sets differ in the rank of the module k, the Centered Binomial Distribution (CBD) parameters η_1, η_2 and the compression parameters d_u, d_v. To speed up the polynomial multiplication, they are first transformed using the NTT before being efficiently multiplied as 128 independent products of degree-one polynomials, and then transformed back into the usual representation using the inverse NTT. A simplified overview of the decapsulation procedure is displayed in Algorithm 1. For brevity and clarity, we have inlined the decryption and re-encryption procedures and modified some variable

names. We refer to the original algorithm specification of Kyber in [ABD+21] for a more detailed explanation of all algorithms and parameters. Throughout the algorithm, bold lowercase letters represent vectors of polynomials, while bold upper case letters represent matrices of polynomials. $\mathcal{B}$ represents the set of all bytes while H denotes SHA3-256.

At the start of the decapsulation, the two parts of the ciphertext are decompressed into $\mathbf{u}$ and v. $\mathbf{u}$ is then transformed into NTT domain and subsequently multiplied with $\hat{\mathbf{s}}$ from the secret key, before the result is transformed back using the inverse NTT, subtracted from v, and then compressed into m', which marks the end of the decryption. The decrypted message m' is hashed together with h into two values, $\bar{K}'$ and r', the latter of which is used to sample $\mathbf{r}$, $\mathbf{e_1}$, and e_2 from a centered binomial distribution using the CBD sampler. The public matrix $\hat{\mathbf{A}}$ is calculated from ρ using the SHAKE-128 Extendable Output Function (XOF). All these values are used to compute $\mathbf{u}'$ and v' that are afterward compared to the original ciphertext. If all values are identical, the correct shared key is derived using $\bar{K}'$. Otherwise, a random key is derived instead using the implicit rejection value z, which is part of the secret key.

3 Adapting the HADES Framework to Arithmetic Masking

In this section, we detail the required extensions to the existing HADES DSE algorithms and to the masking framework which are necessary to efficiently support mixed Boolean and arithmetic masking domains within a single design hierarchy.

3.1 Efficient Pre-Synthesis Critical Path Estimation

The critical path is a crucial performance metric since it directly determines the maximum clock frequency of a hardware design. However, the original version of HADES does not support it as an estimated performance metric for DSE purposes. For Boolean-masked designs like symmetric cryptographic primitives, this is less problematic as the masked non-linear gadgets inherently add register stages throughout the design to stop glitch propagation. Combined with the commonly shallow logic depth of purely linear layers, e.g., MixColumns within AES, this results in reasonably short critical paths, as visible in the initial HADES case studies conducted by Buschkowski et *al.* [BLH+25].

This does not transfer to share-wise field arithmetic which is commonly performed when an arithmetic masking scheme is used. Since no share domains are crossed, registers are optional from a security standpoint. A modular addition or multiplication with a public constant can therefore be implemented purely combinationally unless registers are deliberately placed to break the critical path. For the ML-KEM modulus $q = 3329$, an interleaved modular multiplier chains eleven modular additions (cf. Section 5.2), which leads to a prohibitively long path delay. However, there is also no clear optimal number of register stages to insert since this forms a trade-off between maximum clock frequency on the one hand and latency as well as area on the other hand. To represent this part of the design space, the number of pipeline stages within unmasked templates is treated as another design aspect that can be automatically explored. Since additional registers cost both area and latency without improving any other existing HADES performance metric, all previous optimization targets lead to the minimum number of register stages being inserted. Resulting ML-KEM designs would operate at only 10–20 MHz (cf. Section 7.2).

In order to be able to compete with hand-crafted masked PQC designs operating at clock frequencies of up to several hundred MHz, we extend the HADES framework with the critical path (C) performance metric, as well as derived product metrics like the Area-Critical Path-Product (ACP) which allow for joint optimization of different targets.

Integrating this into the DSE process and evaluating it for every explored configuration requires a way to predict the critical path of a specific design without performing full logic

synthesis and static timing analysis. Naïvely, such a procedure consists of analyzing each path between two register stages and counting the number of logic gates. However, this is not feasible at early stages during the DSE since elaboration of template internals only happens after the performance prediction is done to avoid the involved computational overhead. Therefore, we opt for a two-stage critical path analysis approach instead.

Stage 1: Pre-Selection of the Best Candidates. In order to establish a coarse relation between different configurations of an individual template with regard to the critical path length, we rely on *critical path estimate* formulas provided by template developers. Conceptually, these are similar to the user-defined latency formulas which are already required as part of a HADES template. For a w-bit ripple-carry adder with a configurable number of pipeline stages, such a formula could be

$$\mathsf{C}_{\mathsf{RCA}} = \frac{w}{\mathsf{numPipelineStages}} \cdot \mathsf{C}_{\mathsf{FA}}$$

where C_{FA} corresponds to the critical path estimate of a full-adder sub-template. This is only done once during development of a new template and does not need to be fully accurate. As a result, the added manual workload is limited. To determine the critical path estimate of a full template hierarchy, the maximum estimate across all involved templates is returned.

Such formulas are computationally cheap to evaluate for each explored configuration and can therefore be used to determine the most promising candidate designs during DSE in either exhaustive search or local search mode.

Stage 2: Optional Detailed Analysis of Remaining Designs. After the first stage has already filtered out most of the possible configurations and only the best performing designs according to the given target metric remain, an optional more detailed and therefore also more computationally expensive second stage can be used to re-rank the best candidates (see *detailed prediction* phase in Figure 1).

As part of this phase, HADES fully elaborates the SpinalHDL description of the template hierarchy and walks backwards from every primary output and internal register until a primary input or a preceding register, either explicitly or as a masked AND gadget, is reached, possibly across template boundaries. The length of each path is determined by computing the weighted sum over the number of traversed AND, XOR, and NOT gates. While all gate types are assigned the same weight by default corresponding to a unit delay model, these weights can be replaced with user-defined values depending on the characteristics of the target cell library. The longest encountered path then replaces the critical path estimate from the first stage and is used to output an updated list of best performing configurations.

While we refer to Section 6.2 for a detailed evaluation of the accuracy and performance of both stages, we want to emphasize that this procedure is not meant to accurately predict the maximum clock frequency since this depends on too many aspects which cannot possibly be predicted in a pre-synthesis step, e.g., routing on the target platform or logic optimization performed by the synthesis tool. Instead, it is only supposed to predict how two designs are likely going to perform relative to each other.

3.2 Support for Arithmetic Gadgets

Similar to the existing library of AND, XOR, and NOT gadgets for Boolean masking, arithmetically masked designs in turn require masked MUL, ADD, and NEG gadgets. While the linear functions ADD and NEG operate share-wise, the non-linear secret-times-secret multiplication gadget must be implemented using an arbitrary field multiplication

gadget, e.g., HPC1 or HPC3. Internally, each arithmetic gadget consists of a structural implementation using the existing unmasked Boolean logic gates to construct adders and multipliers.

In order to allow for efficient DSE, the performance prediction logic has also been modified to count these additional gadget types on a per-template basis before accumulating them throughout the full template hierarchy. Area and randomness usage is determined by multiplying the respective counters with the resource utilization per arithmetic gadget, which in turn can be efficiently estimated in a preceding DSE stage by counting the number of Boolean gates in each arithmetic gadget and multiplying it with the pre-computed performance metrics per gate.

3.3 Support for Masking Conversions

When implementing modern PQC primitives, specific parts are usually more efficient to mask using Boolean secret sharing, e.g., symmetric encryption schemes like AES / Rijndael or hash functions like SHA-3 / SHAKE which rely heavily on bitwise operations. On the other hand, operations like polynomial arithmetic are more efficient to mask using an additively arithmetic secret sharing. Therefore, Boolean-to-Arithmetic (B2A) and Arithmetic-to-Boolean (A2B) masking conversions are required to safely transition between different forms of secret sharing. To ensure that the theoretical side-channel security guarantees of the resulting designs are not violated without requiring manual proofs for every use case, we rely exclusively on conversions that adhere to the PINI composability notion, thereby excluding the existing arbitrary-order hardware design by Norga et *al.* [NDKV25]. For our proof-of-concept tool, we implement a modified version of the designs proposed by Bronchain and Cassiers [BC22] which originally target software implementations. Intuitively, the A2B conversion operates by adding all arithmetic shares modulo q using Boolean-masked adders, while the B2A conversion takes d random arithmetic shares $r_0, \ldots, r_{d-1}$, converts them to a Boolean sharing using an A2B conversion, and computes the final output share as $x - \left(\sum_{i=0}^{d-1} r_i\right) \mod q$ where $x \in \mathbb{Z}_q$ is the input to be converted. The Boolean sharing of the final output share can then be safely unmasked before being returned. For additional details, we refer to the original publication.

Since many schemes rely on the conversion of a large number of field elements in immediate succession, e.g., all polynomial coefficients in ML-KEM or ML-DSA, we opt for a fully pipelined hardware architecture, which allows the user to pass a new input to the conversion module in every clock cycle, therefore maximizing throughput while also keeping input scheduling overhead minimal.

As a result, transferring the software reference designs to a hardware implementation consists mostly of instantiating adders of the correct width with pipeline registers in-between in order to ensure all paths are latency-matched. The only security-critical hardware-specific modification of the individual conversions is a single register stage within the B2A design between the IOS mask refresh and the subsequent unmasking step (see [BC22, Algorithm 11, Step 7]) in order to prevent glitch propagation. Due to the inherent sequential nature of software-based execution, this is implicit in the original algorithm description. Since conversions are handled like regular HADES templates during the DSE flow, different types of adders are automatically evaluated, yielding different conversion instantiations depending on the concrete optimization target.

In order to decrease the area requirements in deployment scenarios where both conversion directions are required at different points during algorithm execution, we provide an additional combined conversion template which shares internal resources wherever appropriate. An overview of the resulting hardware architecture is provided in Figure 2. Depending on the mode-select bit `b2a`, either the set of d randomly selected arithmetic output shares or the $d+1$ shares of the conversion input x are passed to the internal A2B

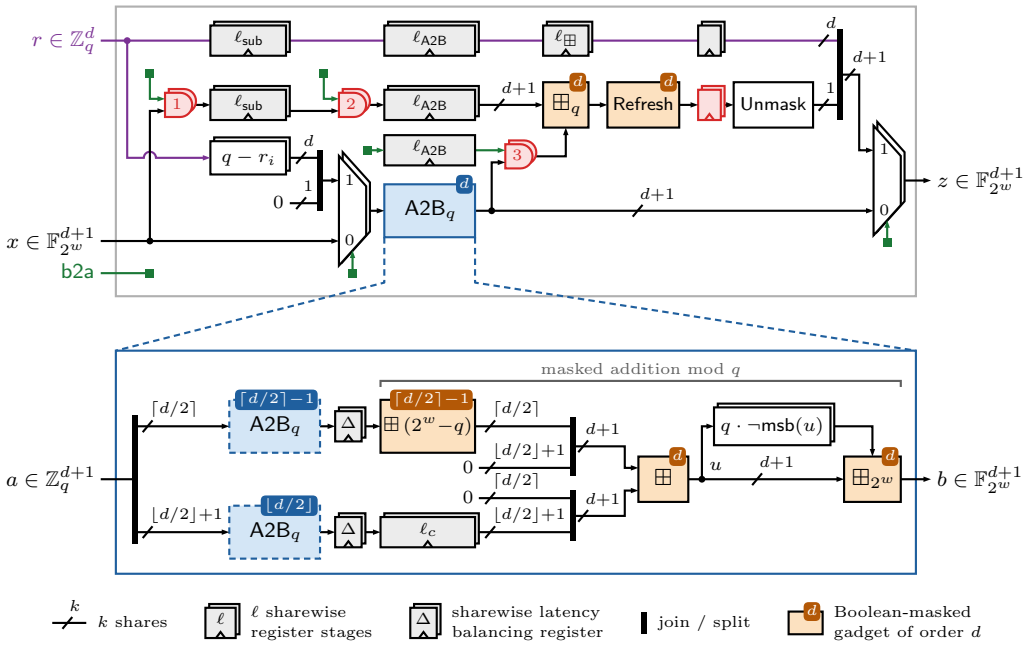


Figure 2. Hardware architecture of our combined B2A and A2B conversion module. Security-critical elements which need to be added to the software-oriented reference design from [BC22] are highlighted in red.

conversion module. When operating in B2A-mode, the result is subsequently post-processed to compute the final non-random output share.

In addition to the previously detailed glitch-stopping register in the B2A data path, several internal signals need to be gated depending on the current mode of operation in order to prevent side-channel leakage, as highlighted within the figure. This form of gating is implemented via a bitwise AND with the mode-select bit on each share individually. As shown by Gigerl et *al.* in [GHP+21], this construction effectively stops glitch propagation assuming that the gate enable bit comes from a stable source, i.e., a register.

More specifically, these AND-gates ensure that the Boolean-masked adder modulo q of the B2A data path only processes data belonging to a B2A operation. For ease of presentation, we analyze the $d = 1$ case in the following to illustrate the underlying problem and the appropriate fixes. Omitting the gating logic, the adder would receive an arithmetic sharing $x_a^{(0)}, x_a^{(1)} \in \mathbb{Z}_q$ with $x_a^{(0)} + x_a^{(1)} \equiv x \mod q$ as the top input in A2B mode, which it would implicitly interpret as a Boolean sharing. The bottom input would meanwhile correspond to a Boolean sharing $x_b^{(0)}, x_b^{(1)} \in \mathbb{F}_2^w$ with $x_b^{(0)} \oplus x_b^{(1)} = x$. The adder would therefore compute $x + (x_a^{(0)} \oplus x_a^{(1)})$. Since $\mathsf{LSB}(x_a^{(0)} \oplus x_a^{(1)}) = \mathsf{LSB}(x_a^{(0)} + x_a^{(1)})$, this is secret-dependent and the subsequent unmasking step would break side-channel security.

This trivially leads to the insertion of gates 2 and 3 as visualized in Figure 2. However, this is insufficient to prevent information leakage in all use cases. When switching from A2B to B2A mode without a full pipeline flush, a subset of the ℓ_{sub} delay registers is going to contain a set of arithmetic sharings of secret inputs. Since the b2a bit is set, gate 2 is going to pass these values through to the adder. Therefore, the sum corresponds to $\left(x_a^{(0)} \oplus x_a^{(1)}\right) + \left(q - \left(\sum_{i=0}^{d-1} r_i \mod q\right)\right)$. Since $\left(x_a^{(0)}, x_a^{(1)}\right)$ is not a valid Boolean sharing of x, the first summand might exceed q and the sum might therefore exceed $2q - 1$. As a result, the conditional subtraction of q within the modular adder might not be sufficient to properly reduce the addition result. The probability of this happening depends on the

secret x, which would again form a potentially exploitable side-channel. Similarly, gates 1 and 3 alone would not be sufficient due to the opposite transition from B2A to A2B mode.

Since the added gates operate purely sharewise, the security properties of the conversion design, including PINI composability, are retained.

In order to assist hardware developers in creating secure mixed-domain-masked designs, each template within HADES is annotated with the type of secret sharing to be used, including the arithmetic modulus if applicable. Unless specified otherwise, HADES automatically analyzes all connections between different templates in a composed hardware design and checks whether source and target masking domains match. If a mismatch is detected, a fully pipelined masking conversion is automatically inserted at that boundary with public signals being automatically delayed by the conversion latency to ensure functional correctness without manual developer intervention.

However, relying on this automatic insertion might introduce unnecessary area overhead in serialized designs. More specifically, if not all cross-domain data paths are active at once, conversions could be shared across different parts of the architecture. To enable such resource sharing within the constraints of our framework, we allow developers to optionally disable automatic conversion insertion. In this case, it is the developers' responsibility to manually insert masking conversions like a regular template and connect them accordingly.

Lastly, we want to emphasize that even though only this specific conversion technique was implemented for our proof-of-concept tool due to its proven PINI composability, future extension of the conversion library is trivially possible and DSE of multiple designs is natively supported.

3.4 Reusing Masked Templates for Public Data

In various PQC schemes, there are operations that are applied to both public, i.e., unshared, and secret, i.e., masked, data at different times of execution. For ML-KEM, one notable sub-module where this applies is the SHA-3 / SHAKE core, which is used both for expanding the public key matrix $\hat{\mathbf{A}}$ but also for hashing the secret decrypted message m' during re-encryption. While the masked ML-KEM hardware implementations presented in [BLH+25] used two separate hashing instances, one unmasked and one masked, the area overhead, especially of the 1600-bit state register, makes it an inefficient solution. To avoid such levels of redundancy, we implement optional support for runtime-configurable unmasked operation to each masked template.

This is achieved by optionally adding a control signal to a template indicating whether public or secret data should be processed. If the signal is set to public, the masking randomness of all gadgets within the corresponding template is set to 0 without the developer manually needing to insert corresponding gating logic. In order to avoid any potential side-channel leakage due to glitches within this randomness multiplexing logic, the select signals are registered. Computation on unmasked values is then implemented by passing the public inputs to the first input share while setting all other shares to 0. As a result, all cross-domain terms within the computation of the corresponding output share cancel out and an unmasked field multiplication is performed. This approach trivially works for all implemented field multiplication gadgets, i.e., HPC1, HPC2, and HPC3, at arbitrary masking orders and in all supported fields.

Even though this technique is effective at reducing area usage of certain masked designs, it also increases latency of public computations since the internal register stages of the masked field multiplication gadgets are not bypassed. While this could theoretically be achieved by adding additional multiplexer paths within each gadget depending on the respective public-vs-secret control signal, this is likely not practical over $\mathbb{F}_2$ since each multiplexer requires at least one AND-gate to realize which could also be used to directly compute the unmasked variant of the AND-gadget.

Table 1. Post-synthesis area and public permutation latency of our d-th order masked Keccak implementation using HPC3 gadgets to perform the χ-step on p slices in parallel with either shared or separate public data path. Data generated using yosys 0.69 targeting NanGate45.

d	p	**Area** [kGE]		**Public latency** [cycles]	
		Shared	Separate	Shared	Separate
0	1	20.3	–	1 600	–
	64	46.6	–	25	–
1	1	47.2	60.7	1 625	24
	4	55.2	67.7	425	24
2	1	72.4	85.8	1 625	24
	4	89.7	100.8	425	24

For the SHA-3 module utilized in ML-KEM, we therefore implemented two variants which are explored using the DSE capabilities of HADES. The first variant utilizes the existing logic within the masked gadgets of the non-linear χ-step for public operations, accepting the resulting latency overhead. The other variant instantiates a separate full-width unmasked round function within the same masked core which is used for operations on public data while reusing the existing state register, thereby allowing for a single-cycle Keccak round with moderate area overhead. A brief performance overview of both resulting core configurations is provided in Table 1.

We further note that this technique can be extended in the future to allow for computation of $d + 1$ unmasked field multiplications in parallel using a single d-th order masked gadget by forcing all cross-domain terms to 0 and supplying one set of inputs to each share domain, as previously done by Mirigaldi et *al.* to speed up Ascon [MPMM26]. For now, this has not been implemented due to open questions with regard to efficient automated scheduling and security implications of the required cross-domain multiplexing logic within a full ML-KEM design.

4 Arithmetic Randomness Generation

While generating random elements from $\mathbb{F}_2$, as required for Boolean masking, is trivial to implement in hardware, the generation of uniformly distributed random elements from an arbitrary field $\mathbb{Z}_q$ for arithmetic masking is significantly more complex. The common approach already utilized in almost all software implementations where non-power-of-2 randomness is required, is rejection sampling. This methodology consists of drawing a bitstring of sufficient length uniformly at random, and then checking whether it is actually a valid field element from $\mathbb{Z}_q$, i.e., if it is smaller than q.

A significant downside of this technique is that sampling becomes a variable-time operation for which the expected execution time depends on the specific modulus. This results in a potential runtime overhead which cannot be prevented in embedded software implementations since sampling and cipher execution typically happen sequentially due to the presence of only one compute unit in most scenarios. In hardware, the situation is different as randomness generation can be performed in parallel to the cryptographic operation itself. However, we still have to take care of rejections accordingly, e.g., by stopping execution of the target algorithm through means of clock gating based on a ready/valid handshake with the randomness generator. Assuming one uniformly random arithmetic element per cycle is consumed, this increases latency by a factor of $1/p$ where p is the acceptance rate of the sampler. For the ML-KEM modulus $q = 3329$, this amounts to a latency increase of 23 %. Furthermore, mechanisms for cipher-wide clock gating or other means of safely stalling masked designs might not always be easily available. As a result, we instead aim for stall-free designs.

The trivial solution to this problem is to simply instantiate more rejection sampling

units in parallel to account for potential rejections and achieve a large enough number of expected valid field elements per cycle to significantly reduce the performance penalty. Even though this decreases the likelihood of a randomness-induced stall, this approach is also quite inefficient as it increases the demand for Boolean randomness per clock cycle with each added unit. Another approach to achieve a lower rejection rate is oversampling, where instead of sampling l bits with minimal l such that $2^l \geq q$, an oversampling factor m is chosen with a corresponding l' such that $2^{l'} \geq m \cdot q$. We now only reject if the sampled value is in the interval $[m \cdot q, 2^{l'} - 1]$ and otherwise apply a reduction modulo q to determine the final sampler output. In addition to requiring more randomness, this also introduces an area overhead due to the involved modular reduction unit.

A complementary approach to further reduce the stalling probability regardless of the configuration of the rejection sampler itself uses FIFO-buffers in order to store excess randomness in cycles where few rejections happen and deploy this stored randomness in cycles where many rejections happen. However, to the best of our knowledge, there has been little to no work regarding potential techniques to make explainable choices for the joint parameters of the sampling instances and the size of the buffer. To solve this issue, we propose a tool-assisted approach which automatically computes the cost-minimal sampling configuration that results in a user-defined failure probability while minimizing the impact on area and latency when implemented in hardware. In the following subsections, we discuss the theoretical foundations and practical implementation of the different steps involved in this process.

4.1 Analysis of Exhaustion Probability

In order to reason about the likelihood of exhausting a randomness buffer of a certain size during algorithm execution, we model its fill-level S as a stochastic process. Each potential fill-level between 0 and the FIFO size F corresponds to one state of the process. In every clock cycle, a constant number of elements n_e is used and thus taken out of the buffer, while a varying number of elements is inserted via the rejection sampling instances. The number of valid rejection sampling outputs trivially follows a binomial distribution $X_r \sim \mathcal{B}(R, p)$, where R refers to the number of parallel rejection sampling instances and p refers to the success rate of a single rejection sampling instance, computed as $p = q/2^l$ for minimum l such that $2^l \geq q$. In case oversampling is applied, the formula for the computation of p changes to $p = (m \cdot q)/2^{l'}$ accordingly, while the rest of the analysis remains applicable. Let S_i refer to the state in clock cycle i, i.e., the fill-level of the buffer. We can determine the probability of transitioning from fill-level k to fill-level j within a single clock cycle as

$$\begin{aligned}\Pr[S_{i+1} = j \mid S_i = k] &= \Pr[X_r - n_e = j - k] = \Pr[X_r = j - k + n_e] \\ &= \binom{R}{j - k + n_e} \cdot p^{j-k+n_e} \cdot (1-p)^{R-(j-k+n_e)}\end{aligned}$$

We additionally define $S_i = 0$ to be an absorbing state, i.e., $\Pr[S_{i+1} = 0 \mid S_i = 0] = 1$ and $\Pr[S_{i+1} = j \mid S_i = 0] = 0 \; \forall j \neq 0$. All transition probabilities can then be combined into a single transition matrix M of size $(F+1) \times (F+1)$. Through exponentiation of this matrix, it is possible to analyze multi-cycle state transitions. More specifically, $(M^{n_c})_{i,j}$ corresponds to the probability of going from fill-level i to fill-level j in n_c clock cycles. By computing $(M^{n_c})_{F,0}$, it is thus possible to determine the probability of exhausting the randomness buffer at some point during the execution of a target algorithm, assuming it requires n_c cycles to run and the buffer is initially completely filled.

Algorithm 2: Parameter exploration for arithmetic randomness generation

Input : Randomness demand per cycle n_e, cipher latency n_c, success probability p, max failure rate ϵ, max buffer size F_{max}, max parallelism R_{max}
Output : Optimal configuration (R^*, F^*) or ERROR

$R \leftarrow 1$; $(R^*, F^*) \leftarrow \perp$; $c^* \leftarrow \infty$
repeat
  improved $\leftarrow$ FALSE
  `// Binary search for the smallest` F `satisfying the failure bound`
  $F \leftarrow 2^{\lceil \log_2 \min(n_e \cdot n_c,\ F_{\mathsf{max}}) \rceil - 1}$
  $s \leftarrow \lfloor F/2 \rfloor$
  while $s > 0$ **do**
    if $\Pr_{\text{fail}}(R, F, n_e, n_c, p) \leq \epsilon$ **then** $F \leftarrow F - s$ **else** $F \leftarrow F + s$
    $s \leftarrow \lfloor s/2 \rfloor$
  end
  if $\Pr_{\text{fail}}(R, F, n_e, n_c, p) > \epsilon$ **then** $F \leftarrow F + 1$
  if $F \leq F_{max}$ ***and*** $\Pr_{\text{fail}}(R, F, n_e, n_c, p) \leq \epsilon$ **then**
    $c \leftarrow \mathsf{Cost}(R, F, p)$
    if $c \leq c^*$ **then**
      $(R^*, F^*, c^*) \leftarrow (R, F, c)$; improved $\leftarrow$ TRUE
    end
  end
  else
    improved $\leftarrow$ TRUE `// keep searching at higher` R
  end
  $R \leftarrow R + 1$
until *no improvement found at parallelism* $R - 1$ ***or*** $R > R_{max}$
if $(R^*, F^*) = \perp$ **then return** ERROR
return (R^*, F^*)

4.2 Automated Exploration of Randomness Generation Parameters

In addition to the exhaustion probability, a cost function parameterized by the number of rejection sampling instances and the buffer size is required in order to effectively determine optimal configurations through a tool-assisted approach. For that, we assign an area cost in Gate Equivalents (GEs) to sampling a single random bit per cycle. While this can be user-defined as part of the HADES configuration file, we set it to $c_{\mathsf{rng}} = 39.4\,\text{GE/(bit/cycle)}$ for our analysis which corresponds to the area requirements of a Trivium instance synthesized using the NanGate45 cell library [HB25]. Similarly, the buffer size is assumed to incur both an area cost, currently estimated at the size of a register, i.e., $c_{\mathsf{reg}} = 4.5\,\text{GE/bit}$, and a latency cost. Generating a single random element through rejection sampling is expected to take $1/(R \cdot p)$ clock cycles on average. Thus, the initial latency required to fill the randomness buffer before cipher execution can be estimated as $F/(R \cdot p)$. Both aspects are combined into a single cost metric using configurable weights w_{area} and w_{latency} as

$$\mathsf{Cost}(R, F, p) = w_{\mathsf{area}} \cdot l \cdot (R \cdot c_{\mathsf{rng}} + F \cdot c_{\mathsf{reg}}) + w_{\mathsf{latency}} \cdot \frac{F}{R \cdot p}$$

where l again refers to the bit width of one sample.

A pseudocode description of the systematic parameter exploration algorithm implemented as part of the modified HADES framework is provided in Algorithm 2. In the

Table 2. Optimal arithmetic randomness generation configurations in terms of sampling parallelism R^* and buffer size F^* for different scenarios with masking order d, latency n_c, and randomness demand n_e, using maximum failure rate $\epsilon = 2^{-128}$ and weights $w_{\mathsf{latency}} = 100 \cdot w_{\mathsf{area}}$.

Scheme	Modulus	Configuration	d	n_c [cycles]	n_e	R^*	F^*	Cost
ML-KEM	3329	**this work**, area-opt.	1	59000	1	4	14	3081.5
ML-KEM	3329	**this work**, latency-opt.	1	18200	1	4	14	3081.5
ML-KEM	3329	**this work**, area-opt.	2	59200	2	5	24	4256.9
ML-KEM	3329	**this work**, latency-opt.	2	18300	2	5	24	4256.9
ML-DSA	8380417	hypothetical design, slow	1	100000	1	2	7	2890.8
ML-DSA	8380417	hypothetical design, fast	1	50000	2	3	9	3954.9

outer loop, we iterate over increasing rejection sampling parallelism degrees. An important observation is that independently of this value, there will be a buffer size which fulfills the allowed exhaustion probability threshold. In the worst case, the size of the buffer is going to correspond to the total randomness demand over the full cipher execution. However, such configurations will most likely not be ideal. In order to efficiently find the minimally required buffer size for each value of R, a binary search technique is utilized. The algorithm terminates once no better, i.e., lower cost, configuration was found for a given rejection sampling parallelism. To bound the execution time, the user is optionally able to define maximum tolerable values for both the buffer size and the rejection sampling parallelism according to the specific requirements of the deployment scenario. If no valid configuration within these bounds could be found, an error is thrown.

We note that this is not part of the main DSE loop itself but an optional post-processing step analyzing how to best implement randomness generation for a specific design configuration which has been identified as a result of the DSE. Estimated area requirements of the randomness generation are then added onto the performance prediction from the preceding DSE step.

To highlight that optimal configurations for randomness generation are often not trivial to determine through human intuition, we provide results of our algorithmic approach for a selection of different parameters in Table 2. The latencies and randomness demands of the ML-KEM scenarios correspond to the masked designs generated as part of this work, while the ML-DSA values are hypothetical figures extrapolated from our results to show how a different choice of modulus with different rejection rate can impact the results. Notably, the area- and latency-optimized ML-KEM designs share the same cost-minimal randomness generation configuration despite latency varying by more than a factor of 3. This is a result of the fact that the buffer size affects the failure probability exponentially while the execution time only has a linear influence.

4.3 Automated Randomness Wrapper Insertion

After determining the optimal sampling configuration given the randomness demand of a (masked) cryptographic template and the user-specified configuration options, a new wrapper template is instantiated surrounding the old toplevel module. Within this randomness wrapper, a corresponding FIFO-buffer is instantiated and rejection sampling is performed. Thus, this new toplevel module only has a single Boolean randomness port which is then internally divided into random bits for the rejection sampling and random bits for potential Boolean-masked sub-templates. As a result, the user no longer has to manually handle any aspects of arithmetic randomness generation, leading to a lower susceptibility to human error. This is particularly valuable since any implementation flaws in these parts of a design can lead to biased masking randomness which quickly compromises the security of a masked circuit. Furthermore, the wrapper automatically

resets the cryptographic target module as soon as a randomness fault occurs, i.e., once more randomness is required than currently present in the internal buffer. This prevents potentially non-deterministic side-channel leakage due to missing masking randomness at some point during cipher execution.

4.4 Handling of Pipelined Designs

One caveat of the previously detailed methodology is that it is specific to serial designs, i.e., designs where only one set of algorithm inputs is processed at once and where start and end points of the execution are clearly defined. For pipelined designs where different executions overlap and, ideally, new inputs are fed into the cipher in each clock cycle, relying on an initial fill-level of the randomness buffer is no longer feasible.

As an alternative solution, we propose implementing periodic checks of the fill-level with user-configurable intervals. The same parameter exploration algorithm can then be applied with n_c corresponding to the check interval instead of the overall target cipher latency. During each checkpoint, the fill-level has to be above some pre-defined minimum. If it is not, the randomness generation wrapper issues a corresponding signal by deasserting the `rand_ready` output. This indicates to the user that the currently available randomness is insufficient for processing additional inputs and that a pipeline stall is required. Once the minimum fill is again reached, `rand_ready` is set and new inputs can be processed. Since our proof-of-concept tool focuses on large-scale PQC ciphers which are likely not fully pipelined, implementation of this modified technique remains as future work.

5 ML-KEM Implementation

To highlight how our extensions to the HADES framework can be utilized to implement an actual PQC scheme, we develop a template implementing ML-KEM decapsulation. As detailed in Section 2.2, this mainly consists of the definition of a configuration-dependent unmasked hardware description which defines the state machine, data path, and memory layout, like in a regular HDL design. We utilize the existing HADES template library where applicable, for example to generate arithmetic modules for use within our ML-KEM Arithmetic Logic Unit (ALU) (see Section 5.2), and extend it where necessary.

Within each sub-template, annotations indicating Boolean or arithmetic masking domains are placed which allow our framework to automatically mask it at arbitrary order. In order to achieve a more efficient implementation, we make use of custom *manually masked components* for the NTT-related templates, including the ALU. This is a newly added feature of HADES that allows circumvention of the regular gadget-based masking flow for certain parts of the design hierarchy. Such a technique enables us to utilize the fact that the NTT is linear over an arithmetic sharing and can therefore be applied as a complete operation on each share individually instead of using masked MUL, NEG, and ADD gadgets for each individual computational step. More specifically, we define a generic NTT butterfly template and create a *manually masked* wrapper around it which automatically instantiates this once per share domain and handles interfacing with other automatically masked templates as well as public control flow that is re-used across all share instances. We want to emphasize that this is a deliberate optimization step which *can* be taken by template developers and not an inherent requirement of our framework. Automated algorithm-agnostic detection of such optimizable linear layers would require tool-based specification-level analysis, which is out of scope for such a DSE framework.

In the following sections, we discuss the general architecture before focusing on a subset of particularly relevant modules in more detail. We implement the decapsulation algorithm as it is the most complex and costly operation, especially with regard to side-channel

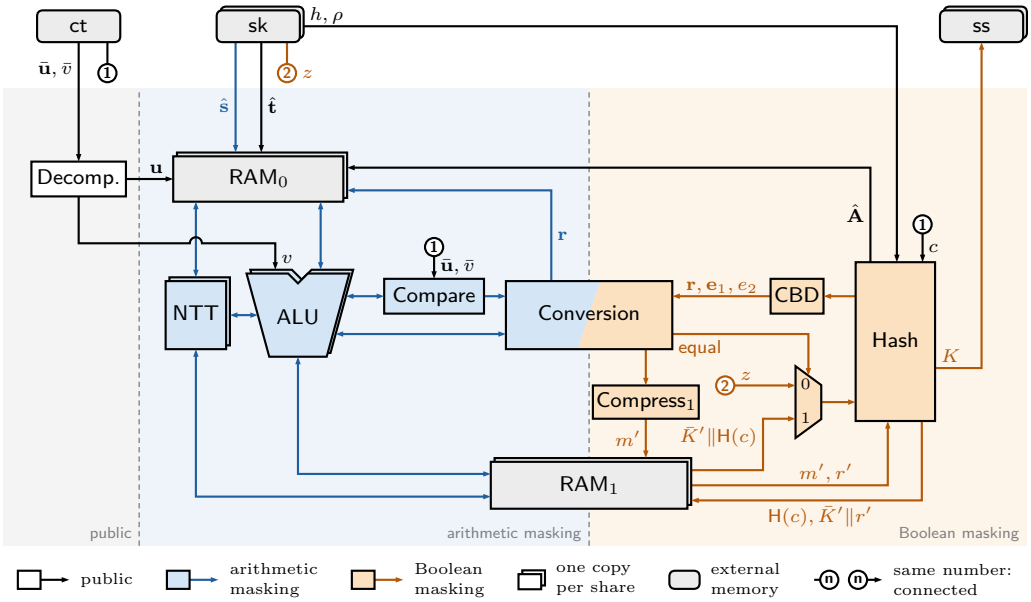


Figure 3. General architecture of the ML-KEM implementation, arranged by masking domain. The bounds of the comparison are computed from public data while the comparison itself happens masked (see Section 5.4).

protection. However, the overall architectural decisions as well as the presented modules are transferable to key generation or encapsulation steps.

5.1 General Architecture

The overall hardware architecture, which is identical for unmasked and masked designs, is shown in Figure 3. We assume that the inputs and outputs of the decapsulation, namely the ciphertext, secret key, and shared secret, are stored in separate, external Random-Access Memory (RAM) blocks. For each of these blocks, we assume a single read or write port respectively with a data width of 64 bit. This allows for easy interaction with the SHA-3 / SHAKE module as it is possible to load or store an entire 64-bit lane of the Keccak state in one clock cycle. Additionally, we use two internal RAMs for storage of intermediate results, namely the two inputs of the polynomial multiplication in RAM_0 and its outputs as well as the different hashed values in RAM_1. Contrary to the external RAMs, the internal RAMs are configured as true dual-port memories which support two address-independent operations with a data width of 32 bit. Using such a configuration, it is still possible for the hash module to process one lane per cycle by accessing two consecutive stored elements. Furthermore, two polynomial coefficients, each requiring 12 bits, can be stored in a single data element, which we utilize in the NTT module and the polynomial multiplication (cf. Section 5.3). If masking is enabled, all internal and external RAM blocks except the public ciphertext input RAM are replicated according to the number of shares, with every share of a secret being stored in a different RAM using the same memory layout. This enables re-use of the unmasked address generation logic independently of the actual masking order.

A central module of our ML-KEM design is the ALU that implements addition, subtraction, and multiplication modulo q, all of which can be executed independently of each other. The ALU is used for all arithmetic operations, especially the NTT and the polynomial multiplication. For the NTT, we use a combined module that can compute both the NTT as well as the inverse NTT, utilizing the ALU for all butterfly operations. We provide

a more detailed description of the ALU and NTT in the following sections. All hashing operations used in ML-KEM are implemented using a single generic hash module, based on the modified Keccak template as described in Section 3.4, that can apply SHA3-256, SHA3-512, SHAKE-128, and SHAKE-256 in both masked and unmasked modes of operation. The masked binomial sampler that operates on the output of the hash module is based on the design proposed in [BGR+21].

The data path crosses masking domain boundaries four times at different stages of the decapsulation procedure (see Algorithm 1). First, an A2B conversion is applied before $\mathsf{Compress}_q$ in line 4. Then, a B2A conversion is used to convert the outputs of the CBD sampler to arithmetic domain in line 7. An additional B2A conversion is utilized for the $\mathsf{Decompress}_q$ operation in line 10 before converting back to Boolean domain using an A2B conversion for the final comparison in lines 11–12.

Since none of these conversions are active simultaneously, we opt against automated insertion of multiple individual conversion modules by HADES and instead manually insert a single combined conversion module as described in Section 3.3, which is then time-multiplexed depending on the current state of execution.

5.2 Arithmetic Logic Unit

As the ALU operates over an arithmetic sharing, it can be masked by replicating each of its arithmetic modules once per share. We note that such simple replication is only possible because all multiplications in ML-KEM happen between one public and one secret operand. If a multiplication of two secret values was used, the ALU would require one of the masked MUL gadgets described in Section 3.2, e.g., HPC1 or HPC3.

The modular addition and subtraction modules in the ALU are either implemented using two adders and a multiplexer to select the right result depending on the carry-bit as described in [BLH+25], or as an interleaved architecture using two Ripple-Carry Adders (RCAs), with the second addition starting as soon as the first bit of the previous addition is available. As a consequence, the result is ready one cycle after the first addition is done instead of requiring two full addition latencies.

For modular multiplication, we use an unrolled interleaved multiplier that consists of eleven stages, each of which doubles the previous partial product of the previous stage and adds one of the two factors using a modular adder if the respective bit of the other one is set. To ensure that multiplication between public and secret data calculates the correct result, the public value has to be multiplied with all secret shares. Thus, the public operand has to be routed to all share inputs of the multiplier, while the secret input is forwarded share-wise.

When targeting Field-Programmable Gate Arrays (FPGAs), advanced primitives like Digital Signal Processors (DSPs) can be used to efficiently perform up to 18-bit multiplication in a single cycle. In order to enable mapping of the ALU to such components, we integrate the option to switch between the usage of Boolean logic gates and a behavioral implementation into the ALU template. If this configuration is selected, the ALU replaces the gate-level modular multiplier with a behavioral multiplication followed by a pipelined Barrett reduction (seven cycles in total), while the gate-level modular adders are replaced with a behavioral addition and a behavioral conditional subtraction. Therefore, in this mode, it has no additional design aspects and a constant critical-path estimate.

5.3 Number-Theoretic Transform

The NTT operates on polynomials stored in RAM and is optimized based on the architecture presented in Section 5.1. It uses a single butterfly unit per share domain and executes all arithmetic using the ALU. One RAM port is used to load the input coefficients needed for each butterfly operation. As two coefficients from (usually) different addresses

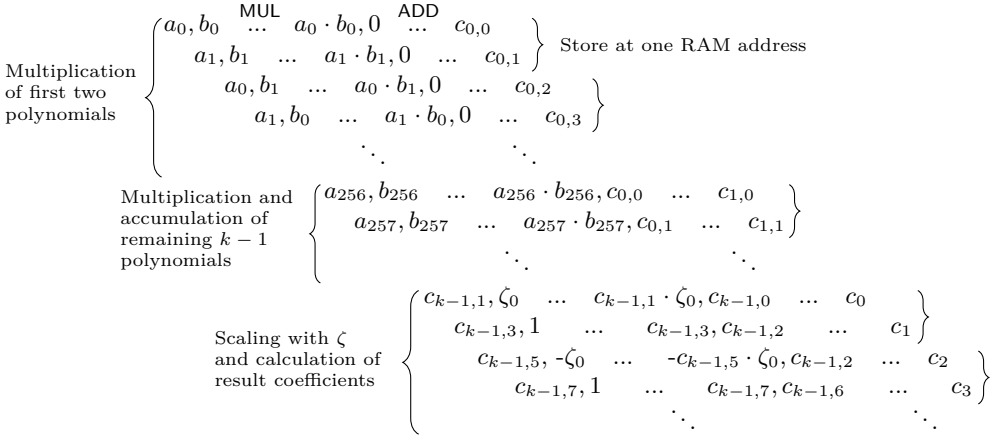


Figure 4. Multiplication of two vectors a and b of polynomials using the multiplier and adder from the ALU. The first polynomial in a vector consists of the coefficients $a_0, ..., a_{255}$ (or $b_0, ..., b_{255}$), the second of $a_{256}, ..., a_{511}$ and so on. Values $c_{i,j}$ are intermediate results, while the coefficients of the result polynomial are $c_0, ..., c_{255}$.

are required, it takes two cycles to load both coefficients. However, since there are two consecutive coefficients stored per RAM address, two pairs of coefficients are available after two load cycles. While these two pairs are being processed by the butterfly unit in the following two cycles, the next two pairs of coefficients are already loaded from RAM, resulting in a fully utilized multiplication pipeline. Once the first results of the butterfly operation are ready, the second RAM port is used to store two consecutive results.

5.3.1 Multiplication of Polynomials in NTT-Domain

As ML-KEM uses an incomplete 7-layer NTT, polynomials still have degree one afterward. Therefore, the multiplication of two polynomials in NTT-domain is not a simple coefficient-wise multiplication, but instead requires 128 multiplications of degree-one polynomials in the form of $(a_0 + a_1X) \cdot (b_0 + b_1X)$. The two resulting coefficients are thus computed as $c_0 = a_0 \cdot b_0 + a_1 \cdot b_1 \cdot \zeta^i$ and $c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$, where $\zeta = 17$ is a 256-th primitive root of unity modulo q and $i = 2 \cdot \mathsf{BitRev}_7(j) + 1$ for the j-th multiplication, i.e., i takes every odd value from 1 to 255. For more information we refer to the ML-KEM specification [ABD+21].

Since ML-KEM operates on vectors of k polynomials, the multiplication procedure not only has to multiply two polynomials, but also needs to accumulate the results of all k polynomial multiplications. The overall structure of the multiplication of two vectors of k polynomials is depicted in Figure 4. This approach attempts to minimize the total number of multiplications while also fully utilizing the multiplication pipeline. For every multiplication of two degree-one polynomials, we first calculate all four possible cross-products of coefficients, i.e., $a_0 \cdot b_0$, $a_0 \cdot b_1$, $a_1 \cdot b_0$, and $a_1 \cdot b_1$. Note that we calculate $a_1 \cdot b_1$ directly after $a_0 \cdot b_0$ as they will contribute to the same output coefficient. Thus, by storing them together in a single 32-bit word, we require only one single-port RAM access when loading both of them during addition. The polynomial multiplication also highlights another advantage of our decision to use internal RAMs with 32 bit data words: with the two available addresses, all data needed to perform a multiplication of degree-one polynomials can be accessed at once. Every result of the multiplier is added to the accumulator, before two successive addition results are stored together in one word in RAM. During the first polynomial multiplication, the accumulator is still zero, but we nonetheless carry out the addition to avoid having two different data paths.

As the scaling with a power of ζ is linear, we do not perform it directly after every

multiplication of two polynomials, but only after all k polynomial multiplications have been accumulated, reducing the total number of required scaling multiplications. In the final step, the accumulated result $c_{k-1,1}$ is multiplied with the corresponding scaling factor and the result is added to $c_{k-1,0}$, which results in the first coefficient c_0 of the output polynomial. All powers of ζ are taken from the twiddle factor Read-Only Memory (ROM) that is also used by the NTT since they are never active at the same time. The value $c_{k-1,3}$ needs no scaling and is thus multiplied with 1 such that it is inserted into the addition pipeline with $c_{k-1,2}$ one cycle after the previous multiplication result.

5.4 Uncompressed Comparison

The final comparison is an integral part of the ML-KEM decapsulation and requires the re-encrypted data to be compressed before comparison with the ciphertext. However, as the compression requires a multiplication followed by rounding, both of which are non-trivial tasks for Boolean as well as arithmetic sharings, we make use of the uncompressed comparison routine that was presented in [BGR+21] and that was also used in the fully Boolean masked ML-KEM implementation in [BLH+25].

For every *public* coefficient from the ciphertext, we first compute the lower and upper bound of the interval that the uncompressed, secret data must be within. Unlike previous works, we do not store these interval boundaries for both compression factors in an external ROM, but instead compute them on the fly from the respective public ciphertext coefficient. Depending on the compression factor, this requires a multiplication with a fixed input as well as additions and shifts, all executed on public data. We present and explain the concrete formulas in Appendix A. This approach requires more internal logic than simply reading the interval boundaries from external ROMs, but it also significantly reduces the storage requirements. More specifically, it avoids a look-up table with one 12-bit bound per possible compressed value, i.e., $(2^{d_u} + 2^{d_v}) \cdot 12$ bits in total, which amounts to 12.5 kbit for ML-KEM-512/768 and 25 kbit for ML-KEM-1024.

To complete the comparison, the interval bounds are subtracted from the shared, uncompressed data using the subtraction module from the ALU. If the data is correct and thus within the computed interval, we expect the subtraction of the lower bound to yield a positive result, while the subtraction of the upper bound should produce a negative result. As the subtractions are done on additive shares, their results have to be converted to Boolean domain first before the borrow-bits, indicating whether the results are positive or negative, can be extracted. These bits are then accumulated with the other correctness bits using a Boolean masked AND-gadget. The final masked multiplexer, choosing between $\bar{K}'$ and z, uses the Boolean-masked accumulated correctness bit as its select signal.

5.5 Other Modules

Sampling from a Centered Binomial Distribution The CBD-sampler receives data from the hash module and treats it as a sequence of 256 values of $2 \cdot \eta$ bits each. Every value is split into two halves, a and b, of equal length whose Hamming weights are subtracted, resulting in one coefficient of the sampled polynomial. The Hamming weights are computed with a masked half adder ($\eta = 2$) or full adder ($\eta = 3$) respectively. As the outputs of the hash module are in Boolean domain and switching between masking domains would be more expensive than a Boolean shared subtraction with small operands, we implement it by adding the bitwise complement of the subtrahend to the minuend using an RCA with the carry-in bit set. Due to the short operand size of only two or three bits, using an RCA is the most area- and latency-effective way of implementing this addition compared to more complex adder structures.

Compression to 1 Bit The compression to 1 bit cannot be replaced by a module similar to the one described in Section 5.4 as the compressed result is needed for future operations. [BGR+21] proposes an efficient implementation of this module, based on the observation that an input between $\lceil \frac{q}{4} \rceil$ and $\lfloor \frac{3q}{4} \rfloor$ is compressed to 1, while all other inputs are compressed to 0. By subtracting $\lceil \frac{q}{4} \rceil$ from the input *before* it is converted from arithmetic to Boolean domain, this interval is shifted such that the compression result is 1 iff the shifted input x lies between 0 and $\frac{q}{2} - 1$. For an input $x = x_{11}x_{10}...x_1x_0$ this can be expressed as $\mathsf{Compress}(x, 1) := \overline{x_{11}} \wedge (\overline{x_{10} \wedge x_9 \wedge (x_8 \vee x_7)})$. This checks that (a) the input is not negative and that (b) the input is smaller than $\lfloor \frac{q}{2} \rfloor$, and can be efficiently implemented using four Boolean-masked AND gadgets and five NOT gadgets.

6 Framework Evaluation

In this section, we evaluate our novel HADES extensions related to mixed-domain masking for several moduli, independently of the ML-KEM case study. This includes analysis of DSE accuracy and performance in Section 6.1, the critical path estimation in Section 6.2, and the masking conversions including their practical side-channel security in Section 6.3.

6.1 DSE Accuracy and Performance

As detailed previously, designs with mixed masking domains, i.e., a combination of Boolean and arithmetic masking, lead to a heavily enlarged design space. This is not only due to the complexity of implemented cryptographic schemes but also due to the configuration options that are present within the basic building blocks, which are mostly atomic in a Boolean setting. Most notably, arithmetic gadgets themselves can contain design aspects. While an HPC1 gadget over $\mathbb{F}_2$ is trivially constructed using AND and XOR gates which directly map to standard cells of the respective technology node, implementation over $\mathbb{Z}_q$ requires modular multipliers and adders to achieve the same functionality, which are themselves explored using the HADES DSE flow. While a behavioral description on RTL level, i.e., using * and % operators in Verilog, would also result in a functionally correct implementation, it would rely on the synthesizer to actually instantiate the internals of these operations. This in turn leads to single-cycle arithmetic circuits which, depending on the bitwidth of the modulus, might fail to meet respective timing constraints, as we will further analyze in Section 6.2.

An overview of the templates that we added or extended for mixed-domain masking, their most important design aspects, and the resulting number of configurations is given in Table 3. As expected, the naïve exhaustive search which still proved to be practically feasible for many symmetric cipher designs in the original version of the framework [BLH+25] remains practical only for individual arithmetic templates, whereas the masking conversions and the complete ML-KEM design require the local search of HADES.

This is evident when looking at DSE runtime per analyzed configuration of a design. On a server with two Intel Xeon Gold 6240 CPUs, evaluating a single configuration of an unmasked modular adder ($q = 3329$) took 0.02 s. The same process for an unmasked ML-KEM decapsulation template required 0.58 s, highlighting how evaluation time scales with template complexity. As a result, performing an exhaustive search over the approximately $5.1 \cdot 10^{18}$ ML-KEM configurations would take roughly 10^{11} years, whereas ten independent local-search runs, each starting from the best of 100 random configurations and optimizing until no further improvement is found, finish after just 25 min in total.

Similarly, it has to be guaranteed that the modified framework still yields accurate performance predictions considering the challenges due to additional gadget types or potential automatic conversion and randomness wrapper insertion which both impact overall design area but are outside of the previous HADES flow. For that, we synthesize a

Table 3. Overview of added or modified (†) templates in order to support mixed-domain masking along with their most important design aspects and the number of overall configurations as estimated by HADES. Boolean and arithmetic gadget families are additional global design aspects.

Template	Design aspects	Configs.
Modular arithmetic:		
Modular adder†	adder type, reduction (sequential/interleaved), pipelining	$1.8 \cdot 10^4$
Modular subtractor	adder type, reduction, pipelining	$1.0 \cdot 10^4$
Interleaved multiplier	adder type, reduction, pipelining of each stage	$3.8 \cdot 10^4$
Building blocks for gadget-based masking:		
HPC1 gadgets[a]	MUL: multiplier, adder, subtractor; ADD, NEG: adder	$1.8 \cdot 10^{19}$
A2B conversion	adder types	$1.9 \cdot 10^3$
B2A conversion	A2B part, masked modular adder, offset adders	$3.0 \cdot 10^7$
Combined A2B/B2A	as B2A (shared A2B part)	$3.0 \cdot 10^7$
Cryptographic components:		
ALU for generic moduli	modular adder, subtractor, multiplier, parallelism	$7.1 \cdot 10^{12}$
Keccak†	parallelism, separate public data path	14
ML-KEM decapsulation	inherited from individual sub-templates	$5.1 \cdot 10^{18}$

[a] MUL, ADD, and NEG are configured jointly as part of the *arithmetic gadget config* within HADES.

selection of arithmetic as well as mixed-domain templates and plot the actual area over the estimated area in Figure 5. Intuitively, the performance prediction outputs are more accurate the closer a point is to the diagonal. We want to highlight that HADES generally over-estimates the area of designs with 77 % of the data points being located below the diagonal and the median ratio between actual and estimated area being 0.82. This is expected considering logic optimization during synthesis which we cannot account for. Overall, 66 % of the designs lie within ±25 % of the HADES estimate.

However, since HADES mainly uses the predictions to sort all evaluated configurations during DSE, we care more about estimating the correct order of designs than estimating the accurate post-synthesis area. In that regard, results indicate that the HADES area metric ranks configurations almost perfectly with a Spearman correlation [Spe04] of $\rho = 0.99$ across $n = 1408$ analyzed designs.

We note that a subset of the accuracy issues which were identified in [BLH+25] still contribute to the observed discrepancies. For example, the area of unmasked multiplexers and public control logic is still only partially included in the area estimation, and closing these gaps is out of scope for this work.

6.2 Critical Path Analysis

To highlight the need for an efficient critical path exploration procedure as implemented in HADES, we provide an overview of maximum clock frequencies generated via synthesis of a behavioral description of the HPC1 multiplication gadget alongside different configurations returned as part of our DSE procedure in Table 4. Results are generated using yosys 0.69 targeting the NanGate45 open cell library. Besides the moduli of ML-KEM and ML-DSA, we include three Mersenne primes with bit widths between 7 and 31 as part of our evaluation.

Both the behavioral description as well as the area-optimal HADES configurations perform the multiplication operation in two clock cycles. As a result, the maximum operating frequency drops from 349 and 313 MHz for a 7-bit modulus to 82 and 46 MHz for a 23-bit modulus, respectively. Synthesis of the 2-cycle HPC1 implementation for the 31-bit modulus did not finish within ten hours. In comparison, HADES configurations optimized for the critical path manage to sustain a maximum clock frequency of at least 1.5 GHz across all moduli, at the cost of 9 to 14 times the area and a latency of 62 to 410 cycles. More balanced designs can be achieved by optimizing for the Area-Critical

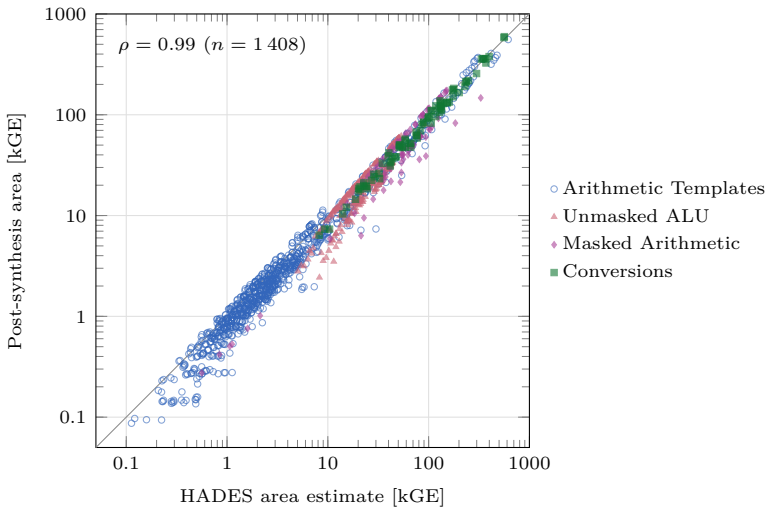


Figure 5. Post-synthesis area using yosys with the NanGate45 cell library compared to HADES area estimate for 1 408 designs, containing: unmasked arithmetic templates (adders, modular adders and subtractors, shift-add-reduce units, multipliers, Decompress), the unmasked ML-KEM ALU, masked arithmetic (ML-KEM ALU at $d = 1, 2$; HPC1 MUL and ADD at $d = 1$–3), and all conversion modules at $d = 1$–3 for the moduli of ML-KEM and ML-DSA. Designs on the diagonal are estimated exactly. ρ corresponds to the Spearman rank correlation.

Table 4. Synthesis results of the first-order HPC1 multiplication gadget for different moduli q using yosys and the NanGate45 cell library. Designs are either based on a naïve 2-cycle behavioral description using in-built arithmetic operators in Verilog or on HADES outputs optimized for A, ACP, and C targets. Maximum frequency is reported based on typical corner, pre-layout OpenSTA reports without wire-load model.

		Behavioral		**HADES A**			**HADES ACP**			**HADES C**		
q	**Bits**	**Area** [kGE]	**Freq.** [MHz]	**Area** [kGE]	**Lat.** [cyc.]	**Freq.** [MHz]	**Area** [kGE]	**Lat.** [cyc.]	**Freq.** [MHz]	**Area** [kGE]	**Lat.** [cyc.]	**Freq.** [MHz]
127	7	3.2	349	3.2	2	313	8.0	11	1401	45.7	80	2388
3 329	12	9.7	183	10.5	2	138	32.1	18	1036	94.7	62	1863
131 071	17	17.8	115	20.2	2	70	42.9	21	1004	177.9	149	1550
8 380 417	23	33.7	82	40.4	2	46	100.0	27	678	499.8	225	1574
2 147 483 647	31	n/a†	n/a†	n/a†	2	n/a†	138.4	38	664	1013.1	410	2206

† Synthesis did not finish within 10 h.

Path-Product (ACP) instead. The resulting configurations can operate at a maximum frequency between 664 and 1 401 MHz at 2.1 to 3.1 times the area of the area-optimized designs and with latencies between 11 and 38 cycles.

In order to further demonstrate that the implemented methodology is effective at accurately estimating the critical path at a pre-synthesis stage, we generated 1 234 unmasked configurations of different templates from the existing library and compared the estimated results after both analysis stages with the actual data as reported by yosys / OpenSTA. For each template, the analyzed configurations are determined by selecting 4 000 random configurations, removing all duplicates, sorting the rest by stage-1 estimate, and extracting up to 200 evenly distributed entries from this list. If a template has fewer than 4000 total configurations, we operate over the exhaustive list of designs instead of using random configuration sampling. The results are visualized in Figure 6 and show a clear correlation between both. To quantify how well the estimates order the set of all configurations by their actual post-synthesis delay, we again rely on the Spearman rank correlation. Notably, the overall Spearman correlation across all tested designs only increases from $\rho = 0.74$ to

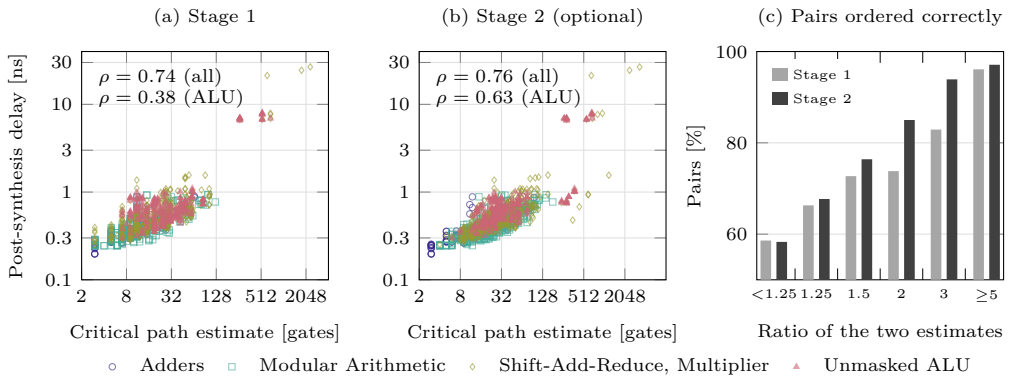


Figure 6. Critical-path estimate compared to post-synthesis delay for 1 234 unmasked configurations generated from the template library and synthesized using yosys with the NanGate45 cell library. Beyond the labeled categories, Keccak, CBD sampler, and Compress templates are included in the evaluation. ρ corresponds to the Spearman rank correlation.

$\rho = 0.76$ when utilizing the optional stage-2 critical path analysis step. However, the real benefit becomes clear when analyzing just the ALU designs. Using just stage-1 estimation, we only achieve a Spearman correlation of $\rho = 0.38$, implying that our estimation fails to consistently find the correct order among these designs. This is due to the fact that the ALU internally consists of the composition of multiple arithmetic sub-templates where the longest path combines across template boundaries. Such cases cannot properly be analyzed without full design elaboration. Once this happens as part of stage 2, correlation increases significantly and reaches $\rho = 0.63$. We want to emphasize that even in such cases, stage 1 still provides enough information to guide the DSE. Even though the exact ordering across structurally similar configurations is not correctly predicted, general trends, e.g., a pipelined design performing better than a purely combinational design, are still captured.

For a set of random configuration pairs from the same design space, Figure 6(c) shows the percentage that are correctly ordered via stage-1 or stage-2 estimation, respectively, depending on the ratio between the estimates for both entries in the pair. As expected, the number of correctly ordered pairs increases with higher ratios since that corresponds to more pronounced critical path differences which allow for larger margins of error during performance prediction, assuming that the estimation broadly correlates with actual delay.

Notably, for very small ratios, i.e., estimates within 25 % of each other, the more detailed analysis is not able to achieve more consistent results, with it only matching the approximately 58.6 % success rate of the stage-1 analysis. For near-ties, the ordering is more likely to change due to logic optimization or other effects during synthesis. Therefore, this result is to be expected. Still, we notice a significant accuracy increase for pairs of configurations with moderately different critical path estimates.

As previously discussed, it is expected that pre-synthesis estimations will not be able to perfectly match actual post-synthesis results since logic optimizations, register balancing, routing congestion, and many other hard to predict properties might influence the resulting critical path. However, the estimate is computationally much cheaper as visualized in Figure 7. Accumulated over all 1 336 analyzed designs (the 1 234 unmasked designs of Figure 6 as well as 93 additional masked ones), the first stage takes 8 min and both stages together 42 min, compared to 8.1 h that yosys and OpenSTA took for the post-synthesis analysis. This corresponds to a performance gain of approximately factor 60 and factor 12, respectively. Considering that the DSE procedure of our ML-KEM template evaluates thousands of configurations per target, the potentially small ordering errors are outweighed by the resulting speed-up.

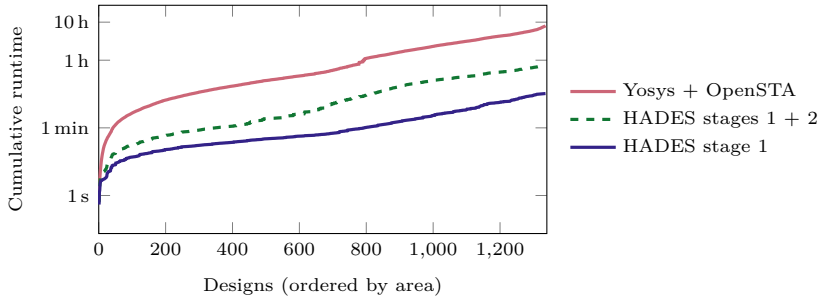


Figure 7. Cumulative runtime of the HADES critical path estimation and of synthesis with timing analysis over all 1 336 synthesized designs of the critical-path evaluation ordered by synthesized area. First stage includes the performance prediction of all other metrics.

Table 5. Post-synthesis area, latency, and randomness metrics of the conversions in their area-optimized HADES configuration for the moduli of ML-KEM and ML-DSA. Latency of the combined conversion is given for A2B and B2A mode separately. Randomness is measured in bits per cycle, where ℓ_q denotes the number of bits required to generate a random element from $\mathbb{Z}_q$.

		A2B			**B2A**			**Combined**		
q	d	**Area** [kGE]	**Lat.** [cyc.]	**Rand.** [bit/cyc.]	**Area** [kGE]	**Lat.** [cyc.]	**Rand.** [bit]	**Area** [kGE]	**Lat.** [cyc.]	**Rand.** [bit/cyc.]
3 329	1	6.4	26	52	17.5	25	$270 + 1 \cdot \ell_q$	18.6	10 / 25	$270 + 1 \cdot \ell_q$
	2	19.5	52	208	48.9	51	$578 + 2 \cdot \ell_q$	50.4	36 / 51	$578 + 2 \cdot \ell_q$
	3	34.2	64	440	83.1	56	$1246 + 3 \cdot \ell_q$	84.6	41 / 56	$1246 + 3 \cdot \ell_q$
8 380 417	1	19.4	48	96	47.6	38	$611 + 1 \cdot \ell_q$	50.0	12 / 38	$611 + 1 \cdot \ell_q$
	2	52.2	60	740	120.5	50	$2262 + 2 \cdot \ell_q$	123.1	24 / 50	$2262 + 2 \cdot \ell_q$
	3	96.0	102	974	210.9	56	$4730 + 3 \cdot \ell_q$	213.8	30 / 56	$4730 + 3 \cdot \ell_q$

6.3 Masking Conversion Handling

Since the B2A and A2B conversions are a crucial part of any mixed-domain masked design, we evaluate all implemented variants, including the combined bidirectional conversion, as part of this section.

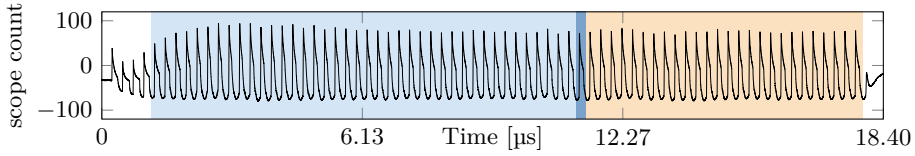
An overview of resource usage and design performance for common PQC moduli is provided in Table 5. Most notably, the combined design requires little additional area compared to a pure B2A implementation, as the B2A conversion inherently already contains a complete A2B conversion. The table further shows an interesting discrepancy. The A2B mode of the combined area-optimized design has a lower latency than the area-optimized standalone A2B configuration, i.e., 10 cycles instead of 26 cycles for $q = 3329$ and $d = 1$. This is due to the fact that multiple values need to be delayed for the A2B latency (see Figure 2), which leads the DSE process to choose a faster parallel-prefix adder instead of the ripple-carry adder in the standalone core. In ML-KEM, one manually inserted combined conversion instance replaces the two B2A and two A2B instances that automated conversion insertion would place (see Section 5.1), thereby reducing overall conversion area from 47.8 to 18.6 kGE and conversion randomness demand from 644 to 270 Boolean bits per cycle at $d = 1$.

In order to validate the practical side-channel security of the resulting designs, we further conduct a set of Test Vector Leakage Assessment (TVLA) measurements [SM15] on a Xilinx FPGA platform of first-order B2A and A2B masking conversions. Such a standalone experiment reduces measurement noise from unrelated components of the design while also allowing us to record a larger number of power traces within a given time frame due to the shorter trace length. More precisely, we analyze a template consisting of a fully pipelined B2A conversion on two 12-bit inputs followed by a share-wise single-cycle

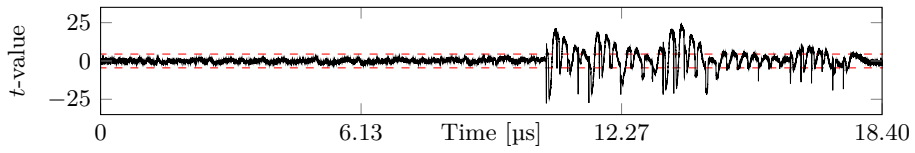
modular addition (for ML-KEM modulus $q = 3329$), and an A2B conversion on the 12-bit result. The conversion instances correspond to the fully pipelined single-direction modules which are used for automated masking conversion insertion within HADES, as described in Section 3.3. We chose the two individual modules for this experiment instead of the combined B2A and A2B design since the latter is already included in the ML-KEM case study and therefore evaluated as part of the full decapsulation analysis in Section 7.1.

The generated Verilog code is synthesized for a SAKURA-G side-channel evaluation board, which is equipped with a Spartan-6 target FPGA, clocked at 4 MHz. The voltage drop over the integrated shunt resistor is amplified using a ZFL-2000GH+ Low Noise Amplifier (LNA) configured with a gain of 4 dB before being digitized using a Spectrum M4 capture card with 8-bit resolution and a sample rate of 1.25 GS/s. In order to generate the required masking randomness, an unrolled Trivium core is instantiated alongside the target design.

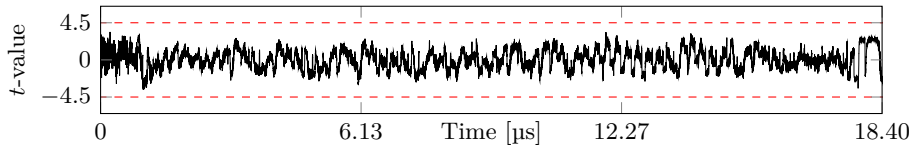
Results of this practical evaluation are provided in Figure 8, including a sample trace (Figure 8a), first-order t-test results for a control measurement using 10 million traces with masking randomness disabled (Figure 8b), and first-order t-test results for the actual leakage assessment measurement with active randomness using 100 million traces (Figure 8c). In order to ensure that the noise level is identical across the mask-off control measurement and the actual evaluation step, the Trivium instance is always active and its outputs are gated using a runtime configuration register. This is required to guarantee that any absence of leakage cannot be attributed to implicit hiding effects due to the additional switching activity of the randomness generator. As expected, with masking randomness disabled, we can observe significant signs of leakage and the commonly used t-value threshold of ± 4.5 is exceeded during both B2A and A2B operations. Since the B2A conversion operates solely on random values for most of its execution, leakage only occurs towards the end. As soon as proper randomness is supplied, no peaks beyond the ± 4.5 threshold are visible.



(a) Sample trace.



(b) First-order t-test using 10 million traces with randomness disabled.



(c) First-order t-test using 100 million traces with randomness enabled.

Figure 8. Side-channel measurement results for a first-order B2A masking conversion (highlighted in light blue), followed by an arithmetically shared single-cycle addition (highlighted in dark blue), and an A2B masking conversion (highlighted in orange).

7 ML-KEM Evaluation

In this section, we evaluate our ML-KEM case study. Results of a practical side-channel security evaluation are presented in Section 7.1 and an extensive performance comparison with unprotected and protected designs from literature is provided in Section 7.2.

7.1 Security Evaluation

We conduct TVLA measurements on both a first-order masked and a second-order masked design optimized for the Area-Critical Path-Latency-Product (ACLP) with the ALU configured in FPGA-mode. Three local search runs with 50 initial configurations each were used to identify the analyzed designs. Since this is a probabilistic procedure, the final configuration varies slightly from the results presented in Table 7. Both designs contain the combined conversion instance and the Keccak core in the shared public data path variant (see Section 3.4). As a result, the measurement implicitly evaluates every possible transition between conversion directions and public / secret hashing. The basic setup is identical to the masking conversion evaluation, except that the target board has been swapped to a SAKURA-X evaluation platform with a larger Kintex-7 FPGA to accommodate the full design, and amplifier gain has been adjusted to 20 dB. Traces are again recorded using a sample rate of 1.25 GS/s leading to a total number of approximately 6.8 million samples per trace. As in Section 6.3, an unrolled Trivium instance with runtime-configurable gating is used for masking randomness generation.

As detailed by Ding et *al.* [DZD+17], the TVLA threshold of ± 4.5 is derived assuming short traces, as is commonly the case for most symmetric algorithms. However, with increasing trace lengths and therefore increasing numbers of sample-wise statistical tests, the Type-I error increases, leading to a high probability of false positives. For traces with more than a million sample points, a threshold of at least ± 6.5 is required in order to reach the intended confidence level. In all following plots, we therefore include both the regular threshold of ± 4.5 as well as the length-adjusted threshold of ± 6.5.

It should be noted that the standard TVLA methodology cannot be as straightforwardly applied as for common symmetric primitives due to the nature of the asymmetric Key Encapsulation Mechanism (KEM). More specifically, testing fixed-vs-random ciphertexts is not feasible since the ciphertext is part of the public data path until combined with parts of the secret key. As a result, the t-test would inherently report leakage during the initial stages of decapsulation, even though there is no dependence on secret data. Simply excluding certain sample ranges of the TVLA results is also not feasible considering that multiple operations are performed in parallel, meaning that these false positives might mask actually exploitable leakage, leading to less meaningful results. While the trivial solution is to analyze sub-components, e.g., the polynomial multiplication, individually, this could also lead to flaws in the complete design remaining undetected. Thus, we instead perform measurements using fixed ciphertexts and fixed-vs-random secret keys which should identify any secret-dependent information leakage throughout the decapsulation. For that, we keep the unshared public parts ρ and h of the secret key constant across both groups. Otherwise, false positives would again occur. The coefficients of $\hat{\mathbf{s}}$, on the other hand, are freshly shared for each trace and correspond to either a fixed value (a key of the known-answer test vectors which accepts the fixed ciphertext) or a random value (generated through rejection sampling modulo q). Therefore, the random group takes the implicit rejection path except with negligible probability. Since this path and the rejection decision are also fully masked, we do not expect this to lead to any side-channel leakage.

Outside of the analysis window, our measurement wrapper verifies that the result memory contains either a sharing of the expected shared secret or of the key derived from the implicit rejection value. All performed decapsulations yielded correct outputs.

Results of this evaluation step are provided in Figure 9 for $d = 1$ and in Figure 10 for

$d = 2$. Each figure contains a reference power trace, an overview of FSM state transitions, first-order t-test results using 100 000 traces with masking randomness disabled, and t-test results up to third order using 10 million traces with randomness enabled. All plots cover the decapsulation procedure from the clock cycle where the enable signal is received to its last clock cycle where the done signal is asserted. With randomness enabled, the first-order t-test reaches maximum values of 5.49 for $d = 1$ and 5.45 for $d = 2$ while the second-order t-test for the $d = 2$ variant reaches 5.21, respectively. Even though these values exceed the threshold of ± 4.5, this cannot be interpreted as signs of leakage. As calculated by Ding et *al.* [DZD+17], even pure noise leads to the threshold being exceeded with probability over 99 % for traces containing over a million sample points. Therefore, following the adapted threshold, these TVLA results indicate that both tested designs meet their expected security targets.

Since our design processes all shares in parallel, univariate leakage starting at analysis order $d + 1$ is expected for a d-th order masked design. This can be observed in the second- and third-order analyses for the $d = 1$ design with peak values of $|t| = 28.97$ and $|t| = 9.54$, which further highlights that our setup is capable of correctly detecting leakage. While we would theoretically expect to see third-order leakage for $d = 2$, the maximum absolute value is only at 3.59. Considering the exponential increase in required traces with higher numbers of shares [CJRR99], this is unsurprising after 10 million traces.

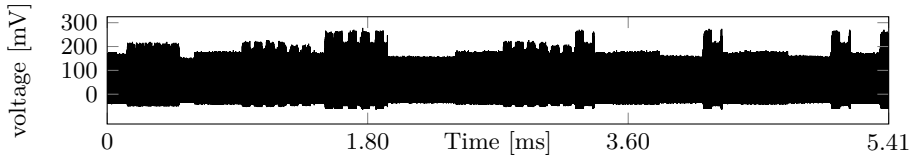
7.2 Performance Evaluation

To evaluate the performance of our proposed ML-KEM architecture in combination with our extensions to HADES, we perform DSE for all three parameter sets of ML-KEM at different security orders. For the sake of brevity, we focus on ML-KEM-512 in the presented tables. As the number of possible configurations reaches billions for an unmasked design and even trillions for masked designs, performing an exhaustive search is infeasible. Instead, we use the local search procedure performing ten independent runs, each starting with an initial sample size of 100 random configurations.

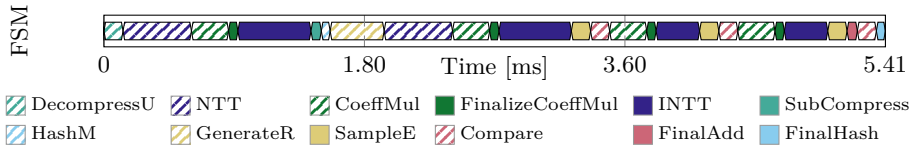
In the following tables, we use various abbreviations: Opt. (optimization target), Rand. (random bits per cycle), and Freq. (maximum clock frequency in MHz). We also use the previously introduced abbreviations for all optimization targets, i.e., A (area), C (critical path), L (latency), and R (randomness), and product metrics built from these.

7.2.1 A Note on HOPE ML-KEM

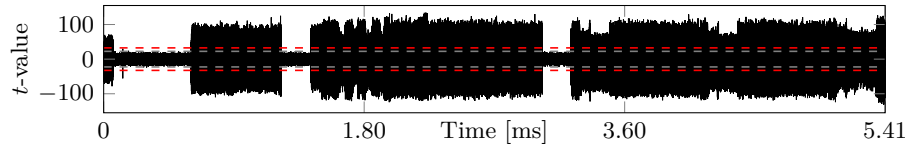
A recently proposed framework specifically tailored to the generation of side-channel protected ML-KEM hardware designs is HOPE ML-KEM [CNC26]. However, their decapsulation implementation differs from ours in terms of masking coverage. While they do claim support for first-order and higher-order masking, this excludes some crucial parts like the CBD sampler which operates on unmasked values and relies on hiding-based protection using Secure Double-Rate Registers (SDRRs) [BBM+18] instead. Similarly, the authors explain that shared values are recombined before compression when operating on values which are assumed to be public. As part of that, the re-encrypted ciphertext within the decapsulation is unmasked, which we verified in their published RTL code. The subsequent comparison with the original ciphertext also happens without additional side-channel protection as the authors argue that their sensitivity analysis from the encapsulation procedure, which correctly flags the ciphertext as a public output, can be transferred to the re-encryption procedure. This is, however, not the case and there are multiple published chosen-ciphertext side-channel attacks that target the re-encryption / comparison operations [BDH+21, UXT+22], which is why other implementations protect this step using masking [OSPG18, DHP+22]. Lastly, we want to point out that they do not include this part in their practical leakage assessment since the evaluation window ends after the



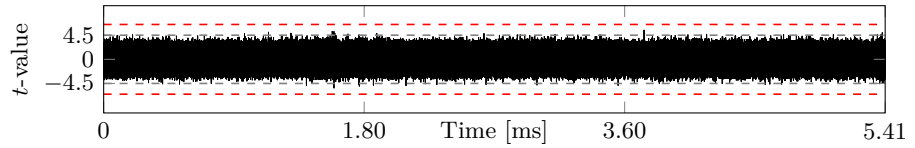
(a) Sample trace.



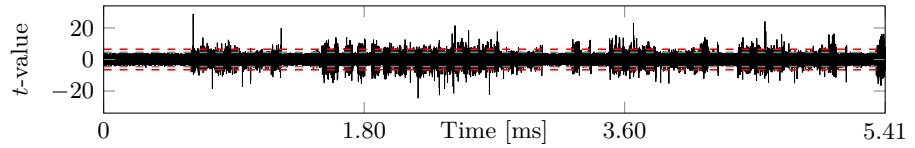
(b) FSM state trace.



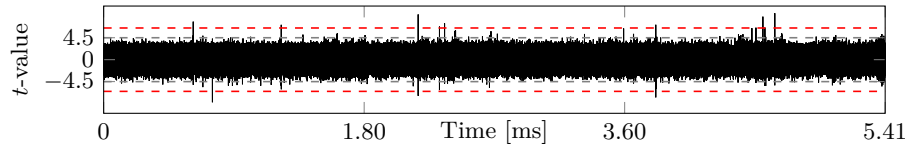
(c) First-order t-test using 100 000 traces with randomness disabled. y-axis uses a logarithmic scale with linear threshold of ±10. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.



(d) First-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.



(e) Second-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.

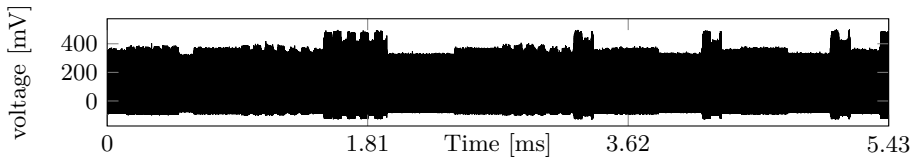


(f) Third-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.

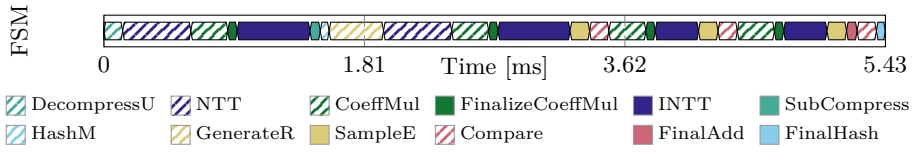
Figure 9. Side-channel measurement results for a first-order masked ML-KEM decapsulation.

IND-CPA decryption is complete, based on the assumption that the re-encryption has the same side-channel properties as the encapsulation.

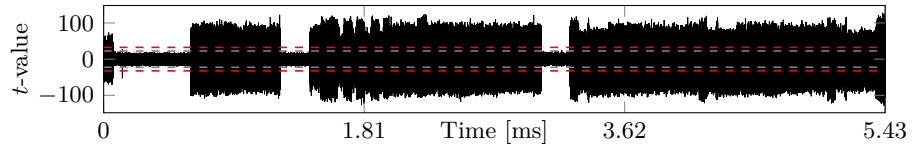
Despite these security concerns, we include performance numbers for their designs in the following tables to capture all relevant literature. Since their framework does not allow for arbitrary masking orders and, more specifically, only supports first-order and third-order masking, their results are not included in the $d = 2$ comparison. We furthermore note



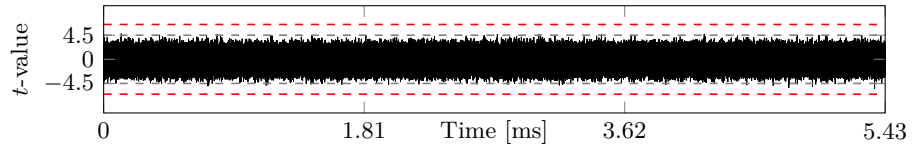
(a) Sample trace.



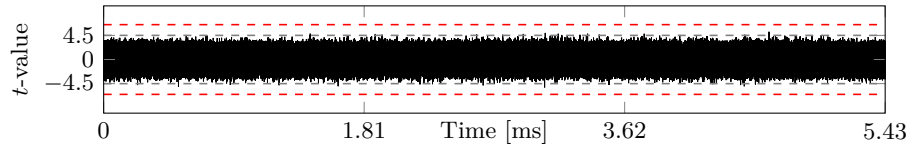
(b) FSM state trace.



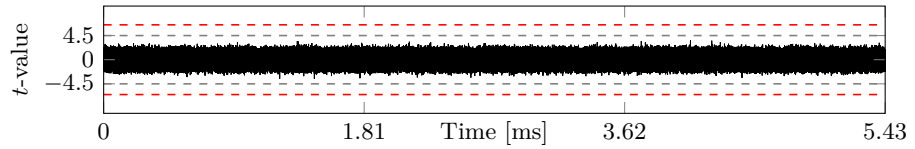
(c) First-order t-test using 100 000 traces with randomness disabled. y-axis uses a logarithmic scale with linear threshold of ±10. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.



(d) First-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.



(e) Second-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.



(f) Third-order t-test using 10 million traces with randomness enabled. TVLA thresholds ±4.5 and ±6.5 in gray and red respectively.

Figure 10. Side-channel measurement results for a second-order masked ML-KEM decapsulation.

that, due to the higher masking coverage of our presented and evaluated ML-KEM designs, additional area and latency overhead are to be expected. Randomness usage cannot be compared since it is not reported by the authors.

7.2.2 ASIC Results

Table 6 shows a selection of the most interesting results of the DSE for Application-Specific Integrated Circuits (ASICs). A collection of all results can be found in Table 9 in Appendix B. Unless explicitly specified otherwise, all designs in these tables target the NanGate45 cell library. To generate the presented data, our designs were synthesized using Synopsys Design Compiler, version S-2021.06-SP4. The provided area metrics exclude all memories and cost of randomness generation. Internal memories are instead accounted for via the separate SRAM column. We always report the average-case latency, taken over 100 executions of the decapsulation with different key pairs and ciphertexts.

In the unmasked case, our designs require less area than all other designs in Table 6, including the ones from [BLH+25]. Compared to the original HADES publication, we achieve lower latencies while also requiring fewer SRAM cells. Furthermore, all designs that consider the critical path during DSE reach a maximum frequency of 427–448 MHz, which is at least 60 % more than the previous HADES designs which target the same cell library. To the best of our knowledge, these results beat all other published designs, except the HOPE ML-KEM outputs, for which the authors report timing closure at 500 MHz.

However, it is also visible that ignoring the critical path during DSE mostly results in implementations with an extremely low maximum clock frequency of less than 20 MHz, which are most likely not suitable for practical use cases, even if the total delay might still be comparable to that of other designs from literature. For these designs, the critical path is located within the interleaved multiplier where all adders are instantiated without internal register stages as well as without registers between adder instances, leading to at least eleven additions being executed between two register stages. These findings demonstrate the importance of the newly implemented critical path metric. Adding the critical path to the area target, i.e., going from A to ACP in Table 9, raises the maximum frequency from 18 MHz to 325–429 MHz at $d = 0$ to 2, while requiring 20–31 % more area, about 1 % more cycles, and keeping the randomness demand unchanged. As a result, this reduces the overall delay by a factor of 18 to 24.

The comparison of masked designs shows similar results as the unmasked case. Again, we manage to achieve significantly lower area while at the same time improving latency. Our A and ACLP designs beat the other first-order masked designs from literature within Table 6 by more than a factor of 2 in area, while losing out to HOPE ML-KEM in terms of latency due to architectural differences which favor higher parallelism. However, while this is a shortcoming of our case study, it also represents another trade-off which could be integrated in the form of an explorable design aspect in the future. Simultaneously, we want to emphasize that not all savings compared to the original HADES designs are due to the different masking technique. Large parts of the gains are due to architectural improvements, e.g., relying on the NTT-based polynomial multiplication instead of Boolean-masked schoolbook multiplication. The benefit of arithmetic masking on the other hand becomes particularly visible when comparing unmasked and masked latencies of different designs. While the latency of the original HADES designs almost doubled when masking is enabled, it increases by only about 11 % for our new ALP optimized designs and by about 2 % for the area-optimized designs. This is due to the fact that no register stages are required inside of the linear arithmetic gadgets. A modular addition which previously required at least 11 cycles [BLH+25] can now theoretically be performed in a single cycle. Lastly, our masked designs whose optimization targets include the critical path reach 325 MHz to 405 MHz, slightly less than in the unmasked case, but still 22–63 % higher than the 249 MHz to 267 MHz for the corresponding Boolean-masked designs.

One small caveat of our designs is the increased SRAM usage for higher security orders. As we require two secret and thus replicated internal RAMs, the SRAM usage grows faster with higher security orders than for the Boolean-masked designs, where most of the SRAM usage is needed for storing the interval boundaries of the comparison, which is independent

Table 6. Selected DSE and synthesis results for ML-KEM-512 decapsulation at different security orders and comparison with previous works targeting ASICs. If not specified otherwise, all designs were synthesized using the NanGate45 cell library. The reported latency is the average-case latency over 100 executions with different data.

Ref.	Opt.	d	Area [kGE]	SRAM [kB]	Rand. [bit]	Lat. [kcycles]	Freq. [MHz]	Delay [µs]
[BSNK19][a,b]	–	0	1340.7	–	–	43.0	200	215.0
[BUC19][a]	–	0	106.0	40.3	–	142.0	72	1972.0
[XHY+20][a,b]	–	0	979.0	12.0	–	80.0	300	266.7
[BAK21][a]	–	0	95.0	10.0	–	10.0	200	50.1
[BLH+25]	A	0	316.0	3.4	–	19 960.1	265	75 321.0
[BLH+25]	ALP	0	459.0	3.4	–	72.0	133	541.0
[CNC26][b,c]	–	0	152.8	–	–	**9.0**	**500**[d]	**18.0**
[this]	A	0	**54.1**	**2.7**	–	58.1	18	3229.8
	ACLP	0	73.6	**2.7**	–	19.6	448	43.7
	ACP	0	70.9	**2.7**	–	58.7	429	136.8
[BLH+25]	ALP	1	668.0	**3.6**	3880	138.4	256	541.0
[BLH+25]	ALRP	1	485.0	**3.6**	211	931.6	249	3741.0
[BLH+25]	R	1	454.0	**3.6**	**163**	52 707.1	256	205 887.0
[CNC26][b,c]	–	1	385.7	–	–	**9.0**	**500**[d]	**18.0**
[this]	A	1	**125.4**	5.4	487	59.0	18	3357.1
	ACLP	1	173.6	5.4	837	22.4	345	65.0
	ACLRP	1	203.8	5.4	214	27.0	376	71.8
	R	1	171.7	5.4	189	60.8	63	966.1
[BLH+25]	ALP	2	1013.0	**3.8**	11 640	138.4	255	543.0
[BLH+25]	ALRP	2	635.0	**3.8**	633	931.6	265	3516.0
[BLH+25]	R	2	590.0	**3.8**	**489**	52 707.1	265	198 895.0
[this]	A	2	**213.2**	8.1	1214	59.2	18	3360.2
	ACLP	2	267.0	8.1	1664	**22.9**	352	**65.0**
	ACLRP	2	344.4	8.1	709	27.8	**405**	68.7
	R	2	517.2	8.1	566	46.1	380	121.2

[a] Technology: not specified [BSNK19], 40 nm [BUC19], 28 nm [XHY+20], 65 nm [BAK21].
[b] Not stated whether the latency is best-case or average-case.
[c] Implements all KEM operations and parameter sets. Area excludes RAM and randomness generation. Uses one butterfly unit per share, processing 4 coefficients at once. At $d = 1$, re-encrypted ciphertext and comparison are unmasked.
[d] Synthesis constraint at which timing closure is claimed; latency from the FPGA implementation.

of the security order. Furthermore, the randomness demand compared to previous designs is mixed. While the new ACLP designs require only about a fifth and a seventh of the randomness that the respective previous ALP designs used for $d = 1$ and $d = 2$, the design with the lowest overall randomness usage still corresponds to a Boolean-masked variant, requiring around 14% less randomness than our best design in this metric. However, this comes at the cost of an increase in latency by a factor of more than 850.

7.2.3 FPGA Results

Table 7 provides post-implementation FPGA results generated using Xilinx Vivado 2024.2 targeting an Artix-7 200T FPGA. The analyzed designs use the adapted FPGA-mode ALU which is optimized for mapping to common FPGA primitives as described in Section 5.2. Notably, this reduces its size per share compared to the gate-level ASIC-mode ALU from 1.1k–2.1k LUTs to only 230–320 LUTs at the cost of 2 additional DSPs. Same as the previous ASICs evaluation, the resource metrics exclude external memories and the cost

Table 7. DSE and synthesis results for ML-KEM-512 decapsulation at different security orders and comparison with previous works targeting FPGAs. If not specified otherwise, all implementations target an Artix-7 200T; ours were implemented with Vivado 2024.2 for speed grade -3. The reported latency is the average-case latency over 100 executions with different data.

Ref.	Opt.	d	LUT [k]	FF [k]	BRAM [units]	DSP [units]	Rand. [bit]	Lat. [kcycles]	Freq. [MHz]	Delay [µs]
[DMG23][b]	–	0	9.5	8.5	4.5	4	–	4.5	220	20.5
[NKK+23]	–	0	16.1	13.9	**2.0**	**0**	–	**2.1**	208	**10.1**
[NDD+25][b,e]	–	0	5.5	3.4	3.5	2	–	6.1	185	33.0
[JGCS24][b,e,f]	–	0	7.2	3.7	5.5	2	–	57.2	**258**	224.0
[CNC26][b,c]	–	0	14.4	10.1	5.0	4	–	9.0	180	50.0
[this]	A	0	**5.2**	2.5	2.5	2	–	58.4	175	333.9
	ACLP	0	6.1	2.6	2.5	2	–	19.1	171	111.4
	L	0	8.9	**2.4**	2.5	2	–	18.4	159	115.6
[KNAH22][b,e]	–	1	152.9	–	489.5	76	–	137.7	100	1377.4
[CNC26][b,c]	–	1	43.5	27.1	7.5	8	–	9.0	**175**	52.9
[DAR+26][e,g]	–	1	42.2	26.1	31.0	8	801	**5.3**	102	**52.3**
[this]	A	1	**12.5**	**6.8**	**4.5**	**4**	**487**	59.2	163	363.7
	ACLP	1	14.8	8.4	**4.5**	**4**	837	22.0	164	133.7
	L	1	31.5	20.7	**4.5**	**4**	3987	18.8	154	121.7
[this]	A	2	22.5	11.1	6.5	6	1214	59.5	165	359.8
	ACLP	2	27.8	13.7	6.5	6	2072	22.3	152	146.4
	L	2	59.5	41.9	6.5	6	11 598	18.9	135	139.7

[b] Not stated whether the latency is best-case or average-case.
[c] Implements all KEM operations and parameter sets. Resources exclude randomness generation. Uses one butterfly unit per share, processing 4 coefficients at once. At $d = 1$, re-encrypted ciphertext and comparison are unmasked.
[e] Device: other Artix-7 [NDD+25, JGCS24], Virtex-7 [KNAH22], Kintex-7 [DAR+26].
[f] Hiding countermeasures only.
[g] Joint ML-KEM and ML-DSA design; resources include ML-DSA support.

of randomness generation. Internal memories are now represented via the BRAM column.

As for the ASIC-mapped designs, our area-optimized designs are, to the best of our knowledge, the most compact unmasked and masked published ML-KEM implementations to date, beating previous work in both LUT count and flip-flop count, while also using at most 0.5 BRAM blocks more than the most memory-efficient design in Table 7. Notably, our area- and ACLP-optimized second-order masked designs need fewer LUTs and FFs than the first-order masked designs of HOPE ML-KEM [CNC26] and of Dobias et *al.* [DAR+26]. However, at the same time, our designs have a higher latency than most other entries in Table 7. Most of this latency difference is caused by the architectural choices within the NTT core, where our latency-optimized designs spend about 70 % of their cycles. This is a consequence of the compact architecture chosen for our case-study, which uses a single butterfly unit per share within the NTT core (see Section 5.3), whereas other designs process more data per clock cycle, e.g., through multiple concurrent butterflies or layer-merged NTT architectures. Since the main contribution of this work is the generic framework and not the concrete ML-KEM implementation, we consider the integration of multiple explorable architectures with varying degrees of data path width to be out of scope. Same as for the ASIC designs, the impact of the masking order on latency is moderate with 15 % overhead at $d = 1$ over the unmasked variant for the ACLP-optimized configuration and 2 % for the area-optimized variant.

In terms of maximum clock frequency, our unmasked designs reach 159 MHz to 175 MHz, which falls behind the 180 MHz to 258 MHz achieved by other unmasked designs from literature. However, our first-order masked designs are about 60 % faster in terms of

maximum clock frequency than the designs of Kamucheka et *al.* [KNAH22] and Dobias et *al.* [DAR+26], while being 6–12 % slower than HOPE ML-KEM. The critical path in the FPGA-mapped designs mostly lies in the control logic, e.g., of the Keccak core or the top-level Finite State Machine (FSM), with routing accounting for 50–78 % of the delay. As these routing effects are significantly more pronounced on an FPGA and cannot be predicted by our pre-synthesis critical path estimation, the C target does not improve the clock frequency further and has thus been omitted from the table.

While our framework is primarily designed for use in ASIC design, these results further highlight that it can also be applied to generate FPGA-mapped designs.

8 Conclusion

In this work, we presented the first generic, security-aware, open-source Electronic Design Automation (EDA) framework that allows for the automated design space exploration and subsequent generation of mixed-domain masked hardware designs of cryptographic schemes. For that, we proposed solutions to three problems that arithmetic masking poses for such a framework and integrated the generic solutions into the HADES tool.

We enhanced the existing DSE capabilities through an efficient two-stage pre-synthesis critical path estimation, which we evaluated for over 1 000 different designs, highlighting that our estimated critical path is accurate enough to guide the DSE process in order to identify the best-performing candidates with regard to this metric. Simultaneously, this procedure requires 12 to 60 times less runtime than traditional post-synthesis static timing analysis. We further enabled the application of the HADES framework to scenarios with mixed-domain masking by extending the gadget library from $\mathbb{F}_2$ to $\mathbb{Z}_q$, adapting the automated masking procedure, and automatically inserting masking conversions at domain boundaries. In addition, template developers can declare manually masked templates and manually place masking conversions, which allows for additional flexibility to implement advanced masking and resource-sharing techniques within the existing constraints of the framework. We also presented a combined bidirectional conversion module, based on the software conversion design by Bronchain and Cassiers [BC22], with hardware-specific fixes to ensure security in all usage scenarios, in particular when switching modes of operation.

Furthermore, we systematically analyzed the sampling and buffering parameters for arithmetic randomness generation by modeling the fill state of the randomness buffer using a stochastic Markov-chain model. As a result, we can accurately and efficiently predict the expected failure rate over the course of a full execution of the cryptographic algorithm, which in turn allows us to determine the optimal configuration for a given target failure probability. We evaluated our proposed methodology for different moduli and integrated the automatic generation of an appropriate randomness wrapper into HADES, which takes care of rejection sampling and buffering without manual developer intervention.

Finally, we evaluated the capabilities of our framework under mixed-domain masking using a newly developed ML-KEM architecture. As part of this process, HADES explored a design space that contains billions to trillions of possible designs, highlighting that the modified framework efficiently scales to the enormous design spaces of modern PQC primitives. While the ML-KEM design itself is not the main focus of this work, the area-optimized configurations are, to the best of our knowledge, the most compact published decapsulation implementations, both masked and unmasked. Fixed-vs-random TVLA measurements on a Xilinx FPGA platform showed no leakage up to the respective masking order for both a first-order masked and a second-order masked decapsulation design after 10 million traces each. Considering that this represents, to our knowledge, the highest published trace count for any first-order ML-KEM decapsulation leakage assessment and the first ever practical evaluation of a complete second-order masked decapsulation design in hardware, we are confident that our resulting designs are actually secure in practice.

With this work, we aim to pave the way for more efficient and secure PQC hardware designs by providing the necessary toolkit to support implementers during the complete development process, reducing the amount of required manual effort both during the conceptualization stage and during the implementation stage.

Appendix

A Interval Bounds for Uncompressed Comparison

ML-KEM compresses the coefficients of $\mathbf{u}'$ to $d = d_u \in \{10, 11\}$ bits and those of v' to $d = d_v \in \{4, 5\}$ bits, depending on the security level [ABD+21]. A public, compressed coefficient $x \in \{0, \ldots, 2^d - 1\}$ matches the secret, uncompressed coefficient w iff $\mathsf{Compress}_q(w, d) = x$. Since the compression routine computes $\lceil (2^d \cdot w)/q \rfloor$, the result equals x iff $(x - 0.5) \leq (2^d \cdot w)/q < (x + 0.5)$. Rearranged, this yields the following bounds for w:

$$\mathsf{Compress}_q(w, d) = x \Leftrightarrow (2x - 1) \cdot q/2^{d+1} \leq w < (2x + 1) \cdot q/2^{d+1}$$

Notably, the upper bound of x corresponds to the lower bound for $x + 1$, thereby enabling the computation of all bounds through a single formula $l(x)$:

$$l(x) = \left\lceil \frac{(2x - 1) \cdot q}{2^{d+1}} \right\rceil \bmod q.$$

For example, for $d = 4$, the input $x = 1$ yields the interval $[105, 313)$, while $x = 0$ yields the interval $[3225, 105)$, wrapping around modulo q. To avoid the modular reduction in hardware, we compute l with the expressions provided below, which we verified exhaustively for all inputs. For $d \in \{4, 5, 10\}$, we first map the input $x = 0$ to $y = 2^d$ while keeping all other values as $y = x$. This mapping yields the same bounds without potentially reaching negative values. All multiplications are by constants and all divisions are by powers of two, so the computation only requires shifts and additions.

Compression to 4 and 5 bits. Due to the fact that $q/2^d$ is close to an integer (208.06 and 104.03, respectively), all intervals can be determined through an affine function of y:

$$l_4(y) = 208 \cdot y - 103, \qquad l_5(y) = 104 \cdot y - 51$$

Compression to 10 and 11 bits. For $d = 10$ and $d = 11$, the interval sizes alternate between 3 and 4 or 1 and 2, respectively, in periodic patterns $[3, 3, 3, 4]$ and $[1, 2, 2, 1, 2, 1, 2, 2]$. However, there are certain exceptions to this pattern since $q/2^d$ (3.2510 and 1.6255) is slightly larger than the average value of each pattern (3.25 and 1.625), resulting in irregular additional interval offsets by 1 position. More specifically, we need to add a correction bit roughly every 256 inputs to account for this mismatch. We define $c_d(y)$ to be 1 iff $y \geq t_i$ and $y \bmod m_d = r_i$ for any pair (t_i, r_i) in Table 8 with $m_{10} = 4$ and $m_{11} = 8$ representing the pattern length. The overall bounds can then be computed as:

$$l_{10}(y) = 3y - 1 + \lfloor (y + 1)/4 \rfloor + c_{10}(y),$$
$$l_{11}(x) = x + \lfloor (x + 1)/4 \rfloor + \lfloor (x + 6)/8 \rfloor + \lfloor (x + 3)/8 \rfloor + \lfloor x/8 \rfloor + c_{11}(x)$$

Table 8. Thresholds t_i and residues r_i of the correction bit c_d for $d = 10$ and $d = 11$.

	i	0	1	2	3	4	5	6	7
$d = 10$	t_i	130	385	644	899				
	r_i	2	1	0	3				
$d = 11$	t_i	134	385	644	903	1154	1413	1672	1923
	r_i	6	1	4	7	2	5	0	3

B Full DSE Results for ML-KEM Targeting ASICs

Table 9 presents an extended version of Table 6 with all DSE results for the ML-KEM-512 instantiation of our case study design, excluding the comparison to related designs from literature for which we refer back to the main body.

Table 9. Full DSE and synthesis results for ML-KEM-512 decapsulation at different security orders targeting ASICs. All designs were synthesized using the NanGate45 cell library. The reported latency is the average-case latency over 100 executions with different data.

Opt.	d	**Area** [kGE]	**SRAM** [kB]	**Rand.** [bit]	**Lat.** [kcycles]	**Freq.** [MHz]	**Delay** [µs]
A	0	54.1	2.7	–	58.1	18	3229.8
ACLP	0	73.6	2.7	–	19.6	448	43.7
ACP	0	70.9	2.7	–	58.7	429	136.8
ALP	0	64.5	2.7	–	19.4	18	1077.7
C	0	109.8	2.7	–	24.6	427	57.6
CLP	0	87.5	2.7	–	18.9	429	44.0
L	0	70.2	2.7	–	17.8	17	1023.0
A	1	125.4	5.4	487	59.0	18	3357.1
ACLP	1	173.6	5.4	837	22.4	345	65.0
ACLRP	1	203.8	5.4	214	27.0	376	71.8
ACP	1	160.0	5.4	487	59.6	325	183.6
ALP	1	143.4	5.4	837	21.5	18	1222.1
ALRP	1	175.4	5.4	214	26.0	18	1470.8
C	1	481.3	5.4	1157	33.9	398	85.1
CLP	1	373.8	5.4	1951	19.6	375	52.3
CRP	1	250.3	5.4	189	63.0	382	165.1
L	1	283.8	5.4	3987	18.2	18	1001.1
R	1	171.7	5.4	189	60.8	63	966.1
A	2	213.2	8.1	1214	59.2	18	3360.2
ACLP	2	267.0	8.1	1664	22.9	352	65.0
ACLRP	2	344.4	8.1	709	27.8	405	68.7
ACP	2	256.7	8.1	1214	59.8	388	154.3
ALP	2	246.1	8.1	1664	21.6	18	1226.0
ALRP	2	312.5	8.1	641	26.5	18	1504.1
C	2	625.1	8.1	6384	31.6	376	84.1
CLP	2	610.9	8.1	11 898	20.0	376	53.1
CRP	2	565.1	8.1	566	46.1	365	126.3
L	2	557.3	8.1	11 786	18.3	12	1522.0
R	2	517.2	8.1	566	46.1	380	121.2

References

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation. 2021.

[AMD+21] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. A Lightweight Implementation of Saber Resistant Against Side-Channel Attacks. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 224–245. Springer, 2021.

[BAA23] Luke Beckwith, Abubakr Abdulgadir, and Reza Azarderakhsh. A Flexible Shared Hardware Accelerator for NIST-Recommended Algorithms CRYSTALS-Kyber and CRYSTALS-Dilithium with SCA Protection. In Mike Rosulek, editor, *Topics in Cryptology - CT-RSA 2023 - Cryptographers' Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*, volume 13871 of *Lecture Notes in Computer Science*, pages 469–490. Springer, 2023.

[BAK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. A Monolithic Hardware Implementation of Kyber: Comparing Apples to Apples in PQC Candidates. In Patrick Longa and Carla Ràfols, editors, *Progress in Cryptology - LATINCRYPT 2021 - 7th International Conference on Cryptology and Information Security in Latin America, Bogotá, Colombia, October 6-8, 2021, Proceedings*, volume 12912 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2021.

[BBM+18] Davide Bellizia, Simone Bongiovanni, Pietro Monsurrò, Giuseppe Scotti, Alessandro Trifiletti, and Francesco Bruno Trotta. Secure Double Rate Registers as an RTL Countermeasure Against Power Analysis Attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 26(7):1368–1376, 2018.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):334–359, 2021.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A Side-Channel-Resistant Implementation of SABER. *ACM J. Emerg. Technol. Comput. Syst.*, 17(2):10:1–10:26, 2021.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and Higher-Order Implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):173–214, 2021.

[BLH+25] Fabian Buschkowski, Georg Land, Niklas Höher, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. HADES: Automated Hardware Design Exploration for Cryptographic Primitives. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(4):1–45, 2025.

[BSG23] Fabian Buschkowski, Pascal Sasdrich, and Tim Güneysu. EasiMask-Towards Efficient, Automated, and Secure Implementation of Masking in Hardware. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023.

[BSNK19] Kanad Basu, Deepraj Soni, Mohammed Nabeel, and Ramesh Karri. NIST Post-Quantum Cryptography- A Hardware Evaluation Study. *IACR Cryptol. ePrint Arch.*, page 47, 2019.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):17–61, 2019.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, Lecture Notes in Computer Science, pages 130–149. Springer, 2015.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, Lecture Notes in Computer Science, pages 188–205. Springer, 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CNC26] Eros Camacho-Ruiz, Pablo Navarro-Torrero, and Alejandro Cabrera Aldaya. A Framework for Designing High-Order Side-Channel Protected Hardware Implementations of ML-KEM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):272–295, April 2026.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[DAR+26] Patrik Dobias, Aikata Aikata, Azade Rezaeezade, Lejla Batina, Lukas Malina, and Tomáš Gerlich. Two Birds, One Mask: Side-Channel-Resistant Unified Hardware for ML-DSA and ML-KEM. *PeerJ Comput. Sci.*, 12:e4003, 2026.

[DHP+22] Jan-Pieter D'Anvers, Daniel Heinz, Peter Pessl, Michiel Van Beirendonck, and Ingrid Verbauwhede. Higher-Order Masked Ciphertext Comparison for Lattice-Based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(2):115–139, 2022.

[DMG23] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. *IEEE Trans. Computers*, 72(2):306–320, 2023.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards Sound and Optimal Leakage Detection Procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, Lecture Notes in Computer Science, pages 105–122. Springer, 2017.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[HB25] Vedad Hadzic and Roderick Bloem. Efficient and Composable Masked AES S-Box Designs Using Optimized Inverters. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):656–683, 2025.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[JGCS24] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A Configurable CRYSTALS-Kyber Hardware Implementation with Side-Channel Protection. *ACM Trans. Embed. Comput. Syst.*, 23(2):33:1–33:25, 2024.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, Lecture Notes in Computer Science, pages 388–397. Springer, 1999.

[KM22] David Knichel and Amir Moradi. Low-Latency Hardware Private Circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1799–1812. ACM, 2022.

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated Generation of Masked Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):589–629, 2022.

[KNAH22] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. A Masked Pure-Hardware Implementation of Kyber Cryptographic Algorithm. In *International Conference on Field-Programmable Technology, (IC)FPT 2022, Hong Kong, December 5-9, 2022*, page 1. IEEE, 2022.

[MPMM26] Mattia Mirigaldi, Nico Paninforni, Maurizio Martina, and Guido Masera. Compact Yet Fast: An Efficient d-Order Masked Implementation of Ascon. In *Design, Automation & Test in Europe Conference, DATE 2026, Verona, Italy, April 20-22, 2026*, pages 1–7. IEEE, 2026.

[Nat24] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS 203, U.S. Department of Commerce, Washington, D.C., August 2024.

[NDD+25] Trong-Hung Nguyen, Duc-Thuan Dam, Phuc-Phan Duong, Binh Kieu-Do-Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang. Efficient Hardware Implementation of the Lightweight CRYSTALS-Kyber. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 72(2):610–622, 2025.

[NDKV25] Quinten Norga, Jan-Pieter D'Anvers, Suparna Kundu, and Ingrid Verbauwhede. X2X: Low-Randomness and High-Throughput A2B and B2A Conversions for d+1 Shares in Hardware. In Matthieu Rivain and Pascal Sasdrich, editors, *Constructive Approaches for Security Analysis and Design of Embedded Systems - First International Conference, CASCADE 2025, Saint-Etienne, France, April 2-4, 2025, Proceedings*, volume 15952 of *Lecture Notes in Computer Science*, pages 119–158. Springer, 2025.

[NKK+23] Ziying Ni, Ayesha Khalid, Dur-e-Shahwar Kundi, Máire O'Neill, and Weiqiang Liu. HPKA: A High-Performance CRYSTALS-Kyber Accelerator Exploring Efficient Pipelining. *IEEE Trans. Computers*, 72(12):3340–3353, 2023.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.

[SH24] Aein Rezaei Shahmirzadi and Michael Hutter. Efficient Boolean-to-Arithmetic Mask Conversion in Hardware. *IACR Commun. Cryptol.*, 1(3):46, 2024.

[Sha79] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, Lecture Notes in Computer Science, pages 495–513. Springer, 2015.

[SPBK26] Nilotpola Sarma, Tapish Patidar, Nupur Brahamanya, and Chandan Karfa. TETRIS: Automated Design Space Exploration of Randomness-Latency Tradeoffs in Masked Hardware. *IACR Cryptol. ePrint Arch.*, 2026:1306, 2026.

[Spe04] C. Spearman. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, Lecture Notes in Computer Science, pages 534–564. Springer, 2019.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 67-I(8):2672–2684, 2020.