HADES: Automated Hardware Design Exploration for Cryptographic Primitives

Fabian Buschkowski[1], Georg Land[1], Jan Richter-Brockmann[1], Pascal Sasdrich[1] and Tim Güneysu[1,2]

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

Abstract. While formal constructions for cryptographic schemes have steadily evolved and emerged over the past decades, the design and implementation of *efficient and secure hardware instances* is still a mostly manual, tedious, and intuition-driven process. With the increasing complexity of modern cryptography, e.g., Post-Quantum Cryptography (PQC) schemes, and consideration of physical implementation attacks, e.g., Side-Channel Analysis (SCA), the design space often grows exorbitantly without developers being able to weigh all design options.

This immediately raises the necessity for tool-assisted Design Space Exploration (DSE) for efficient and secure cryptographic hardware. For this, we present the progressive HADES framework, offering a customizable, extendable, and streamlined DSE for efficient and secure cryptographic hardware accelerators. This tool exhaustively traverses the design space driven by security requirements, rapidly predicts user-defined performance metrics, e.g., area footprint or cycle-accurate latency, and instantiates the most suitable candidate in a synthesizable Hardware Description Language (HDL).

We demonstrate the capabilities of our framework by applying our proof-of-concept implementation to a wide-range selection of state-of-the-art symmetric and PQC schemes, including the ChaCha20 stream cipher and the designated PQC standard Kyber, for which we provide the first set of arbitrary-order masked hardware implementations.

Keywords: Design Automation · Design Space Exploration · Hardware Implementations · High-order Masking · PQC · ML-KEM · ML-DSA · AES · SPN · ARX

1 Introduction

Cryptography is the main pillar for most of our digital security architectures, e.g., the Internet of Things (IoT), that sustain our modern society and are predominantly built and supported by physically accessible constrained and embedded devices. Nowadays, cryptographic primitives are ubiquitously used in hardware and software systems and continuously evolve and emerge through community efforts and standardization competitions, e.g., the CAESAR competition for Authenticated Encryption (AE) [Ber19] (2013 – 2019), or the National Institute of Standards and Technology (NIST) standardization for Post-Quantum Cryptography (PQC) [NIS17] (since 2017). Consequently, continuous generation of *secure* and *efficient* implementations of emerging and existing cryptographic primitives on a wide range of different devices is critical for our modern society to facilitate the use of advanced cryptographic protocols within the security architectures.

Hardware implementations are particularly preferred for most high-assurance and high-performance application scenarios due to their high design flexibility. Specifically, the modular structure of modern (asymmetric) cryptographic primitives, with numerous adaptable sub-components, offers novel possibilities for combination and configuration during implementation. Moreover, the sharing of primitives or their sub-components between various systems and applications greatly expands the design space dimensions.

As a consequence, developers are often unable to accurately assess how design parameters and choices affect the overall efficiency and security of a system. Hence, as design and system complexities increase, *manual* Design Space Exploration (DSE) rapidly becomes infeasible, since optimization for specific constraints turns into a laborious and incremental trial-and-error process, e.g., as demonstrated for the hardware implementations of BIKE [RMG22, RBCGG22] and Dilithium [LSG21, BNG21, ZZW$^+$22].

With this challenge in mind, the ATHENA project [Gaj] for the first time provides a unified and comprehensive Application Programming Interface (API) that enables fair comparison and automated performance number generation for hardware implementations. Unfortunately, as demonstrated during its deployment in standardization competitions, complex design configurations still need to be generated manually by the development teams since no automated exploration process is available. This poses a particular challenge for any team with excellent cryptographic expertise but no hardware background.

Evidently, there is an urgent need for automated tools that assist designers and engineers in implementing and exploring primitives efficiently in hardware. Automating the DSE, i.e., automatically weighing design decisions and predicting efficiency outcomes, can accelerate and optimize the development process and enable a faster deployment of new cryptographic systems and protocols.

However, efficiency is not the only challenge for cryptographic implementations when implemented in hardware and embedded systems. *Physical implementation attacks* are well-known threats to our security architectures, targeting in particular their foundations, i.e., the implementations of cryptographic primitives in hardware and software. More precisely, Side-Channel Analysis (SCA) [KJJ99] enables adversaries to retrieve secret and sensitive information through observation of the behavior and physical characteristics of a device while performing cryptographic operations.

Although sound and effective protection mechanisms, such as masking [CJRR99], have been thoroughly studied in detail and implemented many times, the practical realization of these mechanisms remains a manual, fragile, and error-prone task, regardless of an expert's extensive experience in this field. Only recently, Knichel et *al.* [KMMS22] and Buschkowski et *al.* [BSG23] presented computer-assisted tools that simplify and automate the protection process – however, without support for automated exploration of corresponding design parameters.

Consequently, enabling the automated design space exploration in combination with automated circuit protection in order to find *efficient and secure* hardware instances for cryptographic primitives is an open and urgent research challenge.

Our Contribution

In this work we present a novel approach to model, design, and describe cryptographic hardware, paving the way for automated generation and design space exploration of efficient and secure hardware accelerators. Moreover, we instantiate our concept as a versatile tool which, due to its innovative abstraction and exploration methodology, explores and generates side-channel protected and efficient hardware instances for all kinds of contemporary cryptographic primitives. For this, our tool is particularly designed to rapidly predict user-defined performance metrics during DSE, e.g., area, latency, or randomness quantity, before ultimately instantiating the most suitable candidate in a synthesizable Hardware Description Language (HDL) .

Our modular, extendable, and customizable proof-of-concept implementation of HADES already provides an extensive library of frequently used basic building blocks (called templates) that are required to construct and describe commonly deployed cryptographic primitives. Further, to demonstrate its capabilities, we perform a wide-range case study generating HDL files for efficient and protected symmetric and PQC primitives alike. Eventually, this first comprehensive DSE provides, among others, pioneering results for arbitrary-order masked Application-Specific Integrated Circuits (ASICs) of Add-Rotate-XOR (ARX) ciphers such as ChaCha20 and the designated PQC standard Kyber.

2 Preliminaries

In this section, we briefly introduce the Hardware Construction Language (HCL) SpinalHDL, serving as essential basis for the proof-of-concept implementation of HADES, before we introduce the formal background for our security design, including side-channel countermeasures and protection principles.

2.1 SpinalHDL

SpinalHDL [Spi23] is an HCL embedded into the Object-Oriented Programming (OOP) language Scala. In general, HCLs can describe the functionality of a hardware circuit at a higher abstraction than traditional HDLs such as VHDL or Verilog. For this, HCLs are equipped with powerful libraries and convenience features to ease the process of describing hardware while still having full control over low-level implementation details such as the insertion of registers.

SpinalHDL offers, among other features, dedicated functionalities to describe Finite State Machines (FSMs) and counters. Together with Scala, SpinalHDL offers wide options for the parametrization of designs. In order to integrate into the classical hardware design flow, SpinalHDL provides the necessary translations to both standard VHDL and Verilog, after which the established synthesis tools can be used. Through its black-boxing feature, SpinalHDL is capable of including existing HDL Intellectual Property (IP) into a design by only defining the interface.

2.2 Masking

After Kocher et *al.* presented the first side-channel attack on cryptographic implementations in 1999 [KJJ99], many concepts to protect against this attack vector have been presented over the last two decades. Masking, which is based on Shamir's secret sharing [Sha79], has been established as the most promising countermeasure. To share a secret x, it is split up into $d+1$ shares such that $x = x_0 \circ x_1 \circ ... \circ x_d$. For our work, we consider Boolean masking where the operator $\circ$ is replaced by an `XOR` operation. A correct and secret sharing is achieved by choosing d shares uniform at random (say the first d shares) and computing the remaining share x_d by $x_d = \bigoplus_{i=0}^{d-1} x_i$. Applying this approach ensures that an adversary is not able to learn anything about the secret values by having access to up to d shares.

Current state-of-the-art masked implementations usually follow composability notions, most notably Probe-Isolating Non-Interference (PINI) [CGLS21], which allows the construction of secure circuits from trivially composable atomic units, so-called gadgets. PINI was introduced by Cassiers et *al.* with the concept of Hardware Private Circuit (HPC) and the corresponding HPC1 and HPC2 gadgets. Knichel and Moradi recently proposed HPC3 reducing the latency of the multiplication gadgets to one clock cycle [KM22b].

3 Concept

Implementing hardware modules for cryptographic primitives in constrained applications often requires a rigorous design space exploration to identify optimal or suitable hardware parameters and component configurations that fulfill specified design requirements. With increasing design complexities and sophisticated system architectures, however, it becomes more difficult to precisely determine the interaction of various parameters and configurations, which makes it even more challenging if additional protection against physical implementation attacks is required.

3.1 Problem Definition

To underline the need of new tool-supported concepts for cryptographic hardware design, we start by recapitulating the common practice for developing hardware modules, which commonly includes the following steps:

1. **Specification**: The design goals are defined, considering external constraints such as area demand (cost optimization) and latency (constrained by the module application). For *cryptographic* hardware, additionally, a careful consideration of the adversary model must be carried out, especially regarding side-channel security. This results in additional design goals, namely the targeted level of hardware security.

2. **Functional Design**: The developer writes HDL code to implement the desired functionality, potentially following different approaches (e.g., top-down vs. bottom-up). This process involves making many separate and ad-hoc design decisions – several for symmetric cryptography but even more for asymmetric cryptosystems. Each of those decisions impacts the overall performance metrics, and thus, also whether and how well design goals can be achieved.

3. **Synthesis**: The hardware description is turned into a design implementation in terms of a netlist. Subsequently, the functional correctness and a first post-synthesis area and delay assessment can be carried out, potentially requiring reiterating to the previous step.

4. **Technology Mapping**: Map the synthesized design onto the target technology, ensuring compatibility and efficiency.

5. **Place and Route**: The physical layout of the design on the target device is optimized under consideration of timing, power, and area factors.

6. **Verification and Validation**: In the iterative verification and validation phase, the functional correctness of the final design is tested against diverse input conditions, and the performance of the design is compared to the defined design goals. Simultaneously, the scrutiny of hardware security extends to detecting vulnerabilities by physical measurements but also verification tools. Feedback from each validation iteration is used to adjust the HDL code until the design goals are met, thus requiring an iteration starting from the *second* step.

7. **Deployment.**

With this current workflow that relies on HDL-based tool chains, two main challenges remain: Firstly, whether or not design goals are achieved is determined at the very end of the design process, at least after the first iteration of validation (Step 6). However, we ideally want to be able to estimate whether a design meets its defined performance and hardware security goals *before* synthesis (i.e., between Step 2 and Step 3) to reduce the necessary amount of time-consuming iterations of Steps 3-5. Secondly, due to the lack of security-by-design integration into the tool chains, fulfilling hardware security goals is an error-prone process that leads to more validation iterations and thus, more development costs. Both challenges have their foundation in the fact that the decision-making for design options entirely relies on the developer's experience and intuition, where ideally the workflow should provide tools to enable informed decisions. While this is true for hardware development in general, cryptographic modules having hardware security requirements adds another dimension to the design space, magnifying the complexity of decision consequences.

Consequently, we have identified the following features that are necessary to solve the two aforementioned challenges in the development process, which, however, are not possible with conventional HDLs:

- Integration of security-by-design into the workflow (cf. Section 4.1).
- An automated iteration over different design options *before* synthesis while maintaining functional correctness (cf. Section 4.2.1).
- Prediction of certain performance metrics before synthesis to enable a swift design space traversal given many thousands or millions of design configurations (cf. Section 4.2.2).

3.2 Templates

To address the above stated features, we introduce the concept of *nested hardware templates* as a fundamental principle for abstract modeling and performance prediction of arbitrarily complex hardware circuits for cryptographic primitives. The ensuing abstract and hierarchical hardware model definition particularly enables our DSE concept and the rapid traversal of the search space, as well as the efficient prediction of required performance metrics.

Specifically, our hardware templates are characterized by their extendability and customizability in terms of application-specific algorithm parameters, design-specific configurations as well as the hardware security context, as depicted in Figure 1, enabling prediction of performance characteristics without full instantiation of the system. Using a black-box representation, each template defines the interface (data transmission) and functionality (data processing) of an abstract hardware component. This versatility introduces the ability to address diverse use cases with their unique requirements. Additionally, the hardware security context enables replacing sensitive functionality with secure counterparts.

The implementation of the functionality is covered by a configuration-dependent hardware description (Section 4.1), potentially utilizing various sub-templates. In contrast to modern HDL component representations, e.g., in terms of entities (VHDL) or modules (Verilog), our concept has a hardware security context closely associated, which particularly enables the generation of secure designs in replacing sensitive functionality with secure counterparts. Most importantly, our concept intrinsically enables a *performance prediction* through additional template metadata (Section 3.2.3), which conventional HDLs can only achieve through the integration of new features. This integral capability streamlines the design process by eventually enabling efficient DSE.

In addition to the advances in abstract modeling and performance prediction, our modular approach brings a transformative change by promoting a clear separation between template *designers* and template *users* (Section 4), thereby improving reusability and modularity. Unlike the common practice workflow, where this separation is often implicit and not actively encouraged, our framework promotes a more deliberate division of responsibilities. The template designer, who has specialized knowledge of the intricacies of the hardware and sensitive parts, focuses on creating extensible, customizable, and secure hardware templates with a strong understanding of the various implementation options.

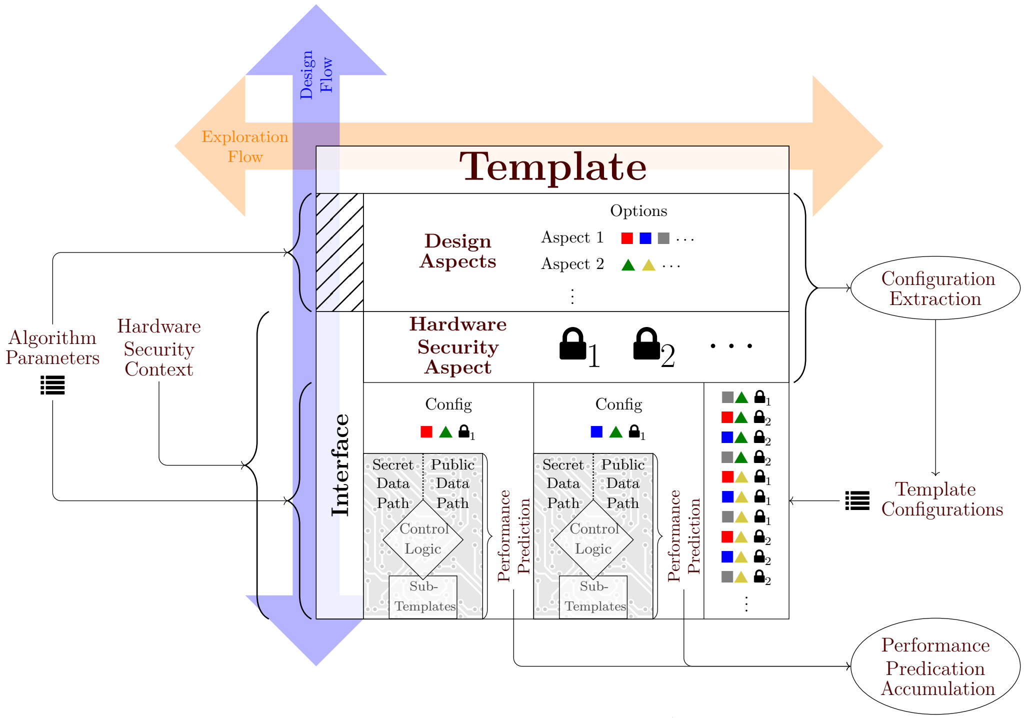


Figure 1. The concept of security-aware hardware templates. As a template can have a different number of subtemplates, depending on the configuration, we refer to this concept also as *nested* hardware templates.

On the other hand, the template user, liberated from concerns about functional correctness of the template or its security features, can seamlessly integrate different options while utilizing the designed templates.

The modular approach presented above promotes cryptoagility and an agile development process in general, while ensuring functional correctness and supporting easy integration of security features. In essence, this moves the concept of hardware development more in the direction of hardware-security-aware, agile cryptographic libraries instead of static modules. This shift is in line with the dynamic, tool-assisted nature of modern development practices and allows for greater adaptability to new cryptographic requirements and facilitates a more responsive and efficient development cycle.

3.2.1 Template Settings

During the process of hardware development, each template in the hierarchy of the hardware model definition is parametrized and configured with three different kinds of settings, as depicted in Figure 1:

Algorithm Parameters are immutable under DSE in defining the inherent algorithmic functionality and external interface of each template in the cryptographic primitive. For hierarchical hardware models and nested templates, this specifically requires algorithm parameter inheritance to extend the parametrization consistently to all child templates. Accordingly, the algorithm parameters mainly define the interface of an instantiated template and its high-level functionality. *Example: The key size of an Advanced Encryption Standard (AES) instance.*

Template Configurations are application-independent and mutable under DSE in solely instantiating template internal data processing mechanisms to realize the requested functionality. Consequently, template configurations do not affect the external template functionality and interface, but exclusively determine how data is processed within a template to ensure consistency throughout the DSE process. Hence, different template configurations have no impact on the interface nor on the high-level functionality. *Examples for hardware addition: Ripple-Carry vs. other adder constructions, unrolled vs. serial, etc.*

The **Hardware Security Context** enables a security-aware design flow. It describes the security level of sensitive data processing or control parts within the template. As such, it might impact the interface as data might be passed and returned redundatly. Similar to the algorithm parameters, the hardware security context is immutable under DSE. *Example: The masking degree to counter side-channel attacks, or other implementation security-related parameters.*

Note that the settings are either application-specific and mutable under the DSE or application-independent and immutable. This strict differentiation facilitates functional correctness, as the template user only defines the application-specific settings and still can be sure to have a functional design after DSE.

This nuanced and highly flexible approach significantly differs from traditional HDLs. In VHDL/Verilog, the parametrization is often limited to static configurations, and even the few dynamic options (e.g., generics) do not offer the degree of flexibility necessary for the different application-specific requirements during DSE. Our approach, on the other hand, not only allows for a more dynamic adjustment of configurations during DSE, but also ensures that the template user's input is focused on application-specific settings, promoting a clearer and more user-friendly design process. The adaptability and granularity of our parametrization and configuration model contribute to improved functional correctness and ease of use, making it a more responsive and efficient alternative to traditional HDLs.

3.2.2 Design Space Exploration

In general, DSE can be employed to achieve the following two goals: (i) finding designs that meet specific performance requirements, or (ii) selecting designs with optimal performance prediction among all possible candidates. While DSE for the first approach may terminate as soon as the predicted performance meets all requirements, the second goal strictly requires exhaustive design space traversal. Consequently, our holistic concept of nested hardware templates is particularly designed to enable both DSE objectives efficiently. In our concept, the preprequisite for a traversal of the entire design space is to extract the full set of design configurations from a template hierarchy. While template configurations are configurations specific to one template, we refer to *design configurations* as the configuration of the entire template hierarchy. We explain the configuration extraction detailedly in Section 4.2. Accordingly, this procedure is applied to the top-level template to perform a full traversal of the design space.

3.2.3 Performance Prediction

Estimation of Performance Metrics (PMs) is a fundamental process to guiding the DSE and evaluating the suitability of different instantiations of the hardware model of a cryptographic primitive with respect to the specified design requirements. For this, each template will predict various user-defined PMs based on its algorithmic parametrization, template configuration, and the hardware security context. For post-configuration performance prediction, these PMs are eventually gathered and accumulated recursively in a bottom-up procedure for each design configuration, i.e., starting with the lowest level before proceeding to the top-level module. Moreover, each template either provides a customized, user-defined aggregation function, or uses simple accumulation as default procedure for the accumulation of PMs.

Furthermore, for appropriate choices of PMs and their estimation, we identified the following requirements and conditions:

Accuracy: The precision and reliability of the PMs prediction is the cornerstone of selecting model instantiations that meet the intended performance goals and targeted requirements.

Efficiency: Estimation of PMs must be fast in order to explore different design options efficiently during DSE. This not only includes computationally efficient prediction routines of PMs but also utilization of minimal data on design characteristics (while still ensuring accuracy).

Coverage: The more hardware module characteristics, e.g., latency, throughput, area utilization, or power consumption, are taken into account, the better a holistic exploration and the consideration of various design trade-offs are possible.

While PMs can generally be chosen and defined by the user, a cycle-accurate latency metric is essential to generate and guarantee functional correctness. In particular, due to our nested hardware template model, modifications to the template configuration of sub-templates may also impact the latency of the parent template. Therefore, cycle-accurate latency prediction is mandatory and must be implemented in any case.

When chosen accordingly, the PMs ideally provide quantitative performance measures while their combination further enables trade-off analysis and balanced design decisions. In contrast to existing manual DSE, which mostly relies on intuition, designer experience, iterative design enhancements, and post-synthesis evaluation, our pre-synthesis and metric-based analysis *objectively* guides design space exploration and candidate selection. Ultimately, pre-synthesis prediction of PMs enables advanced search space traversal strategies alongside micro-optimizations, e.g., in terms of early-stage evaluation and sub-template prioritization or iterative sub-template refinements (see Section 8.3). Particular examples for PMs include, apart from the aforementioned cycle count latency, the area, power consumption, or critical path delay.

4 Workflow

As depicted in Figure 1, we separate between the design and exploration flow. In this section, we present further details of the internals of the nested hardware template concept which, on the one hand, enable the efficient DSE, but on the other hand arise from this separation of design and exploration flow.

We want to stress that our concept is not intended to be a replacement for the standard workflows in hardware development, but rather an extension. With the final output being standard VHDL or Verilog, our proposed workflow can be easily integrated into the established hardware tool chain. In particular, the second step from the common digital design flow ("Functional Design", cf. Section 3.1) is extended in order to reduce the necessary iterations that stem from Step 6, the verification and validation.

4.1 Template Internals

Recall that we differentiate between the template developer[1] and its user. The former defines the behavior of the instantiated hardware module depending on the algorithm parameters, template configuration, and hardware security context. In principle, this

[1] We use the term *designer* synonymously.

process shares some similarities to writing hardware descriptions with generics. As a consequence, experienced hardware developers can easily adapt to our new workflow.

To enable an efficient DSE with functional and security-aware hardware modules as result, the following internals are defined by the template developer.

Design Aspects Depending on the intended functionality, the template designer defines a list of design aspects that represent the freedoms in data processing. Each aspect then has several options that can be selected. For example, the pipelining degree could be a design aspect, with an integer range as the associated options. Since the configuration extraction iterates over these aspect options, they must not interfere with the high-level functionality nor the interface to ensure interoperability after DSE. On the other hand, the algorithm parameters may impact the aspect options, e.g., the available range of unrolling might depend on the AES security level.

Hardware Security Aspect Similar to the design aspects, template designers may define options to instantiate the security-sensitive parts depending on the given security context. For example, if the security context defines a masking degree, there might be different kinds of gadget types to protect the secret data path, each of which has the same masking degree, but different performance characteristics. Then, each gadget type serves as a potential option.

Hardware Description The tuple of selections for (a) the design aspects, and (b) the hardware security aspect is what we call a template configuration. Given a template configuration, a template can be *instantiated*, which means that a specific hardware description is generated. For this, the template designer gives a hardware description relative to each design aspect such that any combination of options can be instantiated. If a combination cannot be instantiated, the designer flags it as a conflict that will be handled by the configuration extraction accordingly.

Performance Prediction Additionally, the template developer gives a generic formula for performance prediction for each metric, which we explain in more detail in Section 4.2. These formulas might depend on the algorithm parameters, hardware security context, and the template configuration.

4.2 Design Space Exploration

As mentioned in Section 3.2.2, the foundation of the DSE is an efficient configuration extraction. This is followed by the performance prediction accumulation, and the actual DSE process is just an iteration over the list of design configurations yielded by the extraction, taking into account the performance prediction as auxiliary information.

4.2.1 Configuration Extraction

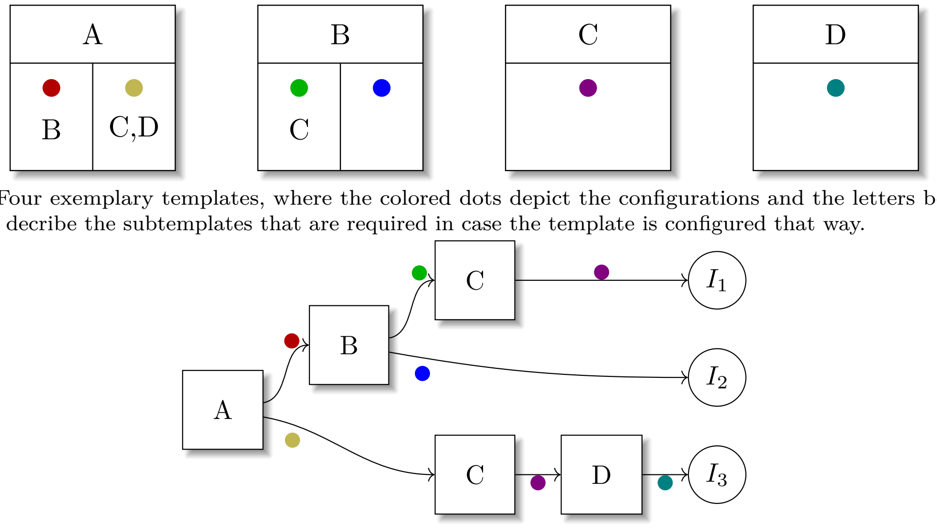
Algorithm 1 presents the details of the configuration extraction procedure. There, a comprehensive list of available template configurations and potential dependencies is extracted for each (sub-)template, as some template configurations may contain mutually exclusive choices of aspect options, yield non-functional designs, or are not yet implemented. This check for conflicts is performed in Line 15.

Essentially, Algorithm 1 traverses a decision tree of configuring templates and their subtemplates. This tree walk is depicted exemplarily in Figure 2. Note how the tree in Figure 2b displays the order in which the templates are configured rather than the template hierarchy, as particularly evident by the fact that D is no subtemplate of C. In fact, the layers of the decision tree are exchangeable to a certain extent: In this example, C and D can be exchanged, since they are independent of each other.

As explained before, continuously ensuring the functional correctness of each design configuration under evaluation requires a strict top-down configuration procedure, which is ensured by the recursion in Algorithm 1. The input template is parametrized with a set of algorithm parameters and a hardware security context. Eventual subtemplates are parametrized with the same hardware security context as it is immutable during DSE. On the other hand, the algorithm parameters of the subtemplates are either chosen statically by the template designer, or derived from the algorithm parameters and the hardware security context of the current template, as shown in Line 11.

Algorithm 1 Configuration extraction procedure

```
 1: procedure ExtractConfigs(template $P_{a,h}$ parametrized with algorithm parameters
    $a$ and hardware security context $h$)
 2:   $C := \emptyset$ (the set of configuration tuples)
 3:   $D :=$ design aspects of $P_{a,h}$
 4:   for all $d \in D$ do
 5:     $O_d :=$ aspect options of $d$
 6:   $H :=$ hardware security options of $P_{a,h}$
 7:   $C_P := H \times O_{d_1} \times \ldots \times O_{d_{|D|}}$            ▷ each element is one config
 8:   for all $c \in C_P$ do
 9:     $S_c :=$ subtemplates of $P_{a,h}$ when configured with $c$
10:     for all $s \in S_c$ do
11:       $a' :=$ DeriveAlgorithmParameters$_c(a, h)$            ▷ defined by designer
12:       $C_s :=$ ExtractConfigs$(s_{a',h})$
13:     $C_c := C_{s_1} \times \ldots \times C_{s_{|S_c|}}$ ▷ each element in $C_c$ is a tuple of subtemplate configura-
        tions
14:     for all $\gamma \in C_c$ do
15:       if $\gamma$ is not flagged as conflict then
16:         extend $\gamma$ with $(P_{a,h}, c)$
17:         $C := C \cup \{\gamma\}$
18:   return $C$
```



(a) Four exemplary templates, where the colored dots depict the configurations and the letters below them decribe the subtemplates that are required in case the template is configured that way.

(b) Decision tree that is traversed by Algorithm 1 when template A from Figure 2a is set as top-level template.

Figure 2. Exemplary depiction of templates and configurations, and how the configurations are extracted by Algorithm 1.

4.2.2 Performance Prediction Accumulation

Based on the list of generated conflict-free design configurations, the actual DSE procedure iteratively selects design configurations and configures the hardware model accordingly. For this, a top-down procedure is applied, ensuring that only those subtemplates are configured that are eventually instantiated given its parents' configuration. Upon completion of design configuration, the templates estimate, accumulate, and report the user-specified PMs in a bottom-up process until the top-level template is reached, yielding the final performance prediction. Depending on the DSE objective, either the current configuration meets the target requirements (i), the DSE procedure continues until a suitable candidate has been found (i), or the design space has been explored exhaustively and the best candidate has been identified (ii). Due to the inherent link between some PMs (e.g., a lower latency often results in a higher area, and vice versa), optimizing all PMs independently is usually not possible. Instead, the DSE optimizes either a single PM, or alternatively a meta-PM that combines multiple PMs, e.g., the product of area and latency.

5 Proof-of-Concept

While Section 3 discusses our general concept for secure and efficient design space exploration and Section 4 presents our proposed workflow, this section briefly introduces and discusses essential aspects of our proof-of-concept implementationof HADES.

5.1 Language Embedding

To implement our proof-of-concept, we require a language or a set of languages that allow us to (i) implement templates with their settings and internals, (ii) express the hardware functionalities of a template, and (iii) perform the DSE.

We opted to follow the OOP paradigm, as it beneficially features the implementation of templates (i) as well as the DSE (iii). In particular, when realizing templates as classes, the template hierarchy can be represented using the instantiation of sub-classes, even when multiple sub-templates of the same type are necessary. Using the inheritance functionality of OOP, algorithm parameters can easily be inherited from the top-level template to its sub-templates. Furthermore, template internals such as the design aspects with their possible options or the PMs can be stored as fields inside the template class. This enables an easy configuration extraction and template configuration during the DSE through dedicated class methods, and allows to access and modify the PM during the performance prediction.

Consequently, we decided to use a Hardware Construction Language embedded into an OOP language to describe a template's functionality (ii). Unlike conventional HDLs that lack essential features for our concept of templates (cf. Section 3.1), HCLs enable an easy hardware description through their rich features and the integration of the hardware description into the respective template class. The OOP language Scala with its embedded HCL, SpinalHDL, is an ideal choice as it is among the most feature-rich and well-maintained HCLs, which makes it easy to learn. In addition, the translations to standard VHDL and Verilog ensures that the generated designs can be further processed by the established tool chain for synthesis and placement. Chisel [Chi23], a second HCL embedded into Scala, was another possible choice. However, unlike Chisel, SpinalHDL (i) is strongly typed, removing possible sources of errors in the template descriptions, and (ii) its black-boxing feature supports generics, which is especially useful when including existing IP with generics into a design.

We also considered the usage of High-Level Synthesis (HLS) to translate a description in a high-level language such as `C` into a hardware description. This approach would allow for a quick and easy implementation of designs through the rich features of high-level languages and require little knowledge about hardware from template designers. However, as high-level languages rely mostly on the HLS tools to optimize the performance of a design, it is often impossible to achieve optimal performances, especially for complex cryptographic designs. Furthermore, certain design aspects such as an unrolling or pipelining factor are often not chosen by the designer, but instead determined by the HLS tools, limiting the amount of different designs that can be explored during DSE. Finally, as high-level languages lack the ability to express some low-level hardware features such as registers, performance prediction prior to HLS would either be very inaccurate, or would require reverse-engineering the HLS algorithms to know how registers are inserted. Given the complexity of current HLS tools, this seems out of reach. Therefore, we deemed HLS to be an unsuitable approach for the hardware description of templates.

5.2 Templates

When implementing templates as Scala classes, we have to ensure that the parametrization settings and template internals are implemented in an appropriate way that matches the requirements from Section 3.2 and Section 4.

To represent their immutability during DSE, algorithm parameters are implemented as constructors to the template class. This also allows to inherit algorithm parameters to potential sub-classes. The hardware security context, which is not only immutable, but also identical for all templates for our proof of concept, is implemented as an object that is created at the start of the DSE and shared between all templates. The design aspects and the hardware security aspect with their possible options are realized as static fields in the template class. This allows the design options to depend on algorithm parameters (cf. Section 4.1) and to access the options during configuration extraction through a dedicated class method. Finally, the PMs for each template are stored in mutable class fields and can be accessed or modified through dedicated class methods.

5.3 Design Space Exploration

The DSE procedure first extracts a list of all possible design configurations according to Algorithm 1 through a recursive extraction method that is called on the top-level template. For every found design configuration, the design hierarchy is configured and the PMs are accumulated in a bottom-up procedure. The PMs of the top-level template, which correspond to the performance of the entire design, are stored in a hash table, using the current design configuration as key and the PMs as value. When sorting this table by a certain PM, it is possible to find the optimal design configuration in the respective category. If multiple design configurations are tied for the best performance, the table can be further sorted by a secondary PM to find the best-suited design.

Our proof-of-concept implementation of HADES currently supports three PMs, namely *Latency*, *Area*, and *Randomness*. While the latency of a template is computed cycle-accurately, the area of a template is estimated based on the amount of registers and logic templates used in a template and a list containing their sizes after synthesis (in Gate Equivalent (GE)). Our implementation is equipped with a list of logic gate and register sizes for the NanGate 45 nm library and can be easily extended for arbitrary cell libraries. The randomness demand of a design can be accurately calculated based on the amount of logic gates used in each template and the current hardware security option. As an unmasked design uses zero bits of randomness, this PM is only viable if a masking degree of one or more is selected. To explore trade-offs between different PMs, the DSE procedure additionally calculates the *Area-Latency Product (ALP)* and *Area-Latency-Randomness Product (ALRP)* by multiplying the respective PMs, yielding designs that are balanced between multiple PMs. Our tool focuses on the trade-offs between area and latency as these are typically the most crucial PMs in hardware development. However, additional trade-offs such as the Area-Randomness Product can easily be added to the DSE process.

6 Case Studies

In this section, we showcase the performance and versatility of our tool and perform DSEs on various common cryptographic components and schemes. Notably, we not only provide implementations of primitives with known implementations in the literature, but also novel implementations, e.g., for ChaCha20 and Kyber, which demonstrates the adoptability of our tool.

For DSE, we aim at an optimization towards five different goals: **Latency**, minimizing the cycle count, **Area**, minimizing the area estimation, **Randomness** (only for masked designs), **Area-Latency-Randomness-Product** (only for masked designs), and **Area-Latency-Product**.

For the synthesis, we use Synopsis Design Compiler, version S-2021.06-SP4, with the NanGate 45 nm library. Moreover, we do not perform an extensive synthesis optimization but rather set the timing goal to 10 µs. Consequently, the reported performance numbers below may be optimized by a more sophisticated synthesis routine.

6.1 Efficient Addition

Efficient addition has seen thorough scrutiny throughout the history of electrical engineering. The simplest method is the Ripple-Carry Adder (RCA), which requires a cascade of n Full Adders (FAs) to add two n-bit numbers. In contrast to this, the group of parallel prefix adders achieves a better critical path delay, which can perform addition with around $\log_2(n)$ stages of so-called propagate-generate groups. Moreover, both types can be implemented in a *serial* or *pipelined* fashion, achieving either a lower area footprint or a higher throughput.

In the field of side-channel security, Schneider et *al.* [SMG15] introduced for the first time arithmetic addition over Boolean-shared values, which makes use of such traditional adder types. Bache and Güneysu [BG22] extended this to higher-order and gadget-based masking. We refer to their works for a comprehensive comparison of different adder types.

Since adders are the base for various cryptographic primitives and higher-level templates, we pre-provide three generic templates with configurable bit width:

1. **Ripple-Carry Adder (RCA)**: This template can be instantiated as a serial or pipelined module (further denoted as sRCA and pRCA). The former works with a single FA that incrementally fills a register stage with its result, while the latter deploys n full adders in parallel. Notably, our pRCA template does not feature a dedicated register stage. Hence, without side-channel protection, the pRCA template has a latency of zero cycles. Only when side-channel protection is activated, the secure gadgets induce a cycle count latency, and thus a fully-pipelined design, which has the same latency as the sRCA but, naturally, a higher throughput.

2. **Sklansky Adder (SKA)**: The SKA has $\log_2(n)$ propagate-generate stages. Analog to the RCA, there are no dedicated register stages in the template, and only with side-channel protection the latency increases to a minimum of $\log_2(n)$ clock cycles.

3. **Kogge-Stone Adder (KSA)**: The KSA has the same number of propagate-generate stages as the SKA but requires more such groups in parallel which, positively, yields a lower fanout.

The template library currently features both SKA and KSA as pipelined versions only. An extension of the adder options is subject to future work.

Table 1 shows the results of the DSE on our adder templates and the synthesis results up to the second masking order. It already highlights several tradeoffs that can be done on a low level. In particular, the masked serial RCA modules are strictly smaller and require less randomness compared to their parallel-prefix counterparts while having a higher latency and delay. Besides, Table 1 shows that the area estimation is often very close to the final area.

Additionally, we provide templates for *modular* addition, which are particularly interesting for more complex use cases like Kyber. For further details and DSE results, we refer to Appendix A.

Table 1. Adder DSE and synthesis results, exemplarily for 16- and 32-bit width. Generally, the adder width is freely configurable.

d	**Opt.**	**Design Config.**		**Area**		**Rand.**	**Lat.**	**Delay**
		gadget	adder	[est. kGE]	[kGE]	[bits]	[cycles]	[ns]
			Algorithm Parameter: 16 bit					
0	L/A/ALP	—	SKA	0.197	0.198	—	0	0.5
1	L/ALP	HPC3	SKA	8.8	6.5	116	5	1.8
	A	HPC3	sRCA	1.4	1.3	4	16	42.0
	R/ALRP	HPC2	sRCA	1.5	1.4	2	32	55.5
2	L/ALP	HPC3	SKA	18.9	13.4	348	5	2.1
	A	HPC3	sRCA	3.1	2.1	12	16	33.7
	R/ALRP	HPC2	sRCA	3.3	2.2	6	32	55.5
			Algorithm Parameter: 32 bit					
0	L/A/ALP	—	SKA	0.481	0.482	—	0	0.7
1	L/ALP	HPC3	SKA	20.8	16.7	304	6	2.6
	A	HPC3	sRCA	2.5	2.4	4	32	73.9
	R/ALRP	HPC2	sRCA	2.6	2.5	2	64	107.8
2	L/ALP	HPC3	SKA	48.2	34.8	912	6	3.4
	A	HPC3	sRCA	5.6	3.6	12	32	82.5
	R/ALRP	HPC2	sRCA	5.8	3.8	6	64	140.2

6.2 ARX Ciphers

Using only the adder templates, we can already securely instantiate a whole family of cryptographic ciphers consisting only of addition, rotation, and XOR. Notably, the latter two operations can be performed share-wise for PINI gadgets such as the HPC gadgets. Exemplarily, we have implemented a template for ChaCha20, configurable for the **adder type** (including serial vs pipelined), **number of quarter rounds** to be performed in parallel (1, 2, or 4), and **number of adders** for the final addition of input and updated state (1, 2, 4, 8, or 16).

Table 2 shows the result of our DSE and the synthesis. The number of adders in the final addition only has a minor influence on the latency, but a major influence on area and randomness. This becomes obvious when comparing the optimizations for latency and ALP, which only differ in the number of adders used. While the latency optimization only improves the latency by 13 %, the area increases by almost 200 %.

6.3 Keccak

The Keccak family of permutation functions can be used differently to instantiate hash functions or Extendible Output Functions (XOFs), most notably SHA3 variations and SHAKE. Masking is especially required for use cases in which Keccak processes secret data, for example within Kyber.

Table 2. ChaCha20 DSE and synthesis results for up to the third masking degree.

d	**Opt.**	**Design Config.**				**Area**		**Rand.**	**Lat.**	**Delay**
		# QR	# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
0	L	4	16	SKA	—	24	43	—	236	0.34
	A	1	1	SKA	—	13	29	—	611	0.96
	ALP	4	2	SKA	—	18	39	—	243	0.38
1	L	4	16	SKA	HPC3	475	396	6 080	722	1.04
	A	1	1	sRCA	HPC3	55	53	8	22 787	30.55
	R	1	1	sRCA	HPC2	55	53	4	33 539	45.94
	ALRP	2	1	pRCA	HPC2	61	61	6	17 619	26.06
	ALP	4	1	SKA	HPC3	163	147	1 520	827	1.12
2	L	4	16	SKA	HPC3	1 096	786	18 240	722	1.13
	A	1	1	sRCA	HPC3	123	76	24	22 787	42.12
	R	1	1	sRCA	HPC2	124	77	12	33 539	46.26
	ALRP	2	1	pRCA	HPC2	137	88	18	17 619	29.41
	ALP	4	1	SKA	HPC3	374	266	4 560	827	1.21

The permutation consists of five steps: the first three (θ, ρ, π) are linear in the Boolean masking domain, while the χ step is a quadratic function and requires non-linear gates, and the final ι is an affine function. The main parameter to explore during DSE, particularly for masked designs, is the number of parallel χ operations performed. Currently, our template is restricted to Keccak-f[1600] – a generalization is planned as future work – and it supports 25, 50, 100, 200, 400, 800, or 1600 parallel χ operations. During each permutation iteration, the first three steps θ, ρ, π are carried out simultaneously within a single clock cycle, and the outcome is then stored in the state register. Subsequently, the χ step is performed with the configured parallelism, and the ι step is performed in a final clock cycle. This procedure is repeated 24 times, as given by the Keccak specification.

For masked designs, most area and all randomness is spent on the quadratic χ step, which is the only one to require (costly) non-linear gadgets. Hence, a tradeoff exists between low latency (many χ operations in parallel) and low area and randomness (few χ operations). The tradeoff is noticeable in the results of the DSE shown in Table 3.

Table 3. Performance results for Keccak-f[1600], using different optimization targets for up to third masking degree. Latency and delay are given for one permutation consisting of 24 rounds.

d	**Opt.**	**Design Config.**		**Area**		**Rand.**	**Lat.**	**Delay**
		para.	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[ns]
0	L	64	—	12.9	22.0	—	96	30.72
	A	1	—	6.7	16.2	—	1 608	916.76
	ALP	32	—	9.8	19.3	—	120	115.16
1	L	64	HPC3	131.6	149.4	3 200	120	40.80
	A	1	HPC3	15.1	30.1	50	3 144	1 792.47
	R/ARLP	1	HPC2	15.9	34.6	25	4 680	3 744.00
	ALP	16	HPC3	42.8	61.6	800	264	198.05
2	L	64	HPC3	299.3	325.9	9 600	120	40.80
	A	1	HPC3	24.3	52.2	150	3 144	2 954.89
	R/ARLP	1	HPC2	26.2	54.1	75	4 680	2 901.43
	ALP	16	HPC3	89.8	117.7	2 400	264	211.20

6.4 AES

AES is one of the most widely used symmetric encryption algorithms, and there are various implementation strategies with different performance characteristics. Therefore, finding the best-suited implementation among all possible designs is of crucial importance, especially when also considering security against physical attacks.

Our AES template that supports both encryption and decryption is parametrizable for the key size, namely for 128, 192, and 256 bit keys. During DSE, the following design aspects are explored: The architecture (round-based or unrolled), the types of S-Boxes for the round function, and the key schedule (we currently support the Canright S-Box and the Boyar-Peralta S-Box, and different choices for the round and the key schedule are possible), and the number of parallel S-Boxes (1, 2, 4, 8, or 16) and MixColumns instances (1, 2, or 4) in a round. Table 4 shows the results of the DSE. Most notably, at first and second order, the unrolled architecture is chosen when optimizing for latency. Due to the inherent register stages of masked non-linear gadgets, this architecture only improves the latency by 8 % over the round-based ALP optimization, while increasing the area by over 600 %.

Table 4. DSE and synthesis results for AES-128. The chosen design configurations and further results for AES-192 and AES-256 can be found in Table 14.

d	**Opt.**	**Area**		**Rand.**	**Latency**	**Delay**
		[est. kGE]	[kGE]	[bit]	[cycles]	[ns]
0	L/ALP	17.3	32.9	—	15	501
	A	4.4	9.9	—	656	1 292
1	L	814.4	860.8	11 592	51	83
	A	20.0	23.8	144	1 616	1 972
	R/ALRP	22.1	26.1	68	2 576	3 503
	ALP	119.5	125.9	1 224	55	100
2	L	1 819.3	1 670.8	34 764	51	115
	A	44.1	40.0	408	1 616	2 101
	R/ALRP	49.2	45.2	204	2 576	3 812
	ALP	267.3	227.0	3 660	55	116

6.5 Polynomial Multiplication

Recently, Land et *al.* [LMRG24] showcased the feasibility of applying gadget-based masking to modern Public-Key Cryptography (PKC), and particularly lattice-based PQC schemes, by implementing Streamlined NTRU Prime completely with HPC2 gadgets. They observe that polynomial multiplications, as often required by lattice-based schemes, are feasible with Boolean masking if there is a public operand and the secret operand has only a small number of possible coefficient values. In this case, it is possible to deploy Schoolbook multiplication. This approach boils down to

1. multiplying the coefficient of the public operand by each potential secret coefficient value (this can be done without side-channel protection), then
2. securely multiplexing each of these public values with the secret coefficient as the "select" input, and finally
3. securely accumulating the mux result using an adder.

The downside of this approach is that implementations of most PQC schemes usually deploy dedicated multiplication routines like NTT or Karatsuba, which are algorithmically faster but infeasible to be used with Boolean masking due to the big intermediate coefficient multiplications.

The template we implement follows the Schoolbook strategy to enable masking easily and features configurability regarding the reduction polynomial, coefficient modulus, adder types, and the number of adders that are instantiated in parallel. Notably, the list of potential numbers of parallel adders is computed individually for each reduction polynomial, such that the search complexity of the DSE remains reasonable. Table 5 contains the DSE results for the polynomial multiplication for the Kyber-512 use case.

Sparse Multiplication

We extend the above-explained concept to another relevant case: sparse multiplication. The use case for this is the signing procedure of Dilithium (cf. [LMRG24, Sec. 6.5]), the designated PQC signature standard. Dilithium performs a sparse multiplication which is highly vulnerable to side-channel analysis, as the so-called challenge polynomial c is multiplied with the secret key $\mathbf{s}_1, \mathbf{s}_2$, and subsequently added to the signature nonce $\mathbf{y}$. In this case, c is sparse and public as pointed out in [KLRBG23, ABC+23], and the secret key is a secret polynomial vector with small coefficients bounded by $\eta \in \{2, 4\}$. We observe that – similar to the concept above – no coefficient multiplications between "big" integers are required.

The template for our sparse multiplication is configurable for the reduction polynomial (but currently only supports polynomials with the form $x^n \pm 1$), the sparsity, the range of the secret. The template configuration consists of the adder type and the number of adders that are instantiated. For further implementation details as well as DSE and synthesis, we refer to Appendix B.

Table 5. DSE and synthesis results for polynomial multiplication. Exemplarily, we use the Kyber reduction polynomial $X^{256} + 1$ with the modulus $q = 3329$ and a secret range $\eta = 3$ (Kyber-512). Further results can be found in Appendix D (Table 15).

d	Opt.	Design Config.			Area		Rand.	Latency	Delay
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
0	L	256	SKA	—	450	n/s*	—	1 281	n/s*
	A	1	sRCA	—	40	94	—	2 490 625	4 408
	ALP	43	SKA	—	108	1 192	—	2 561	16
1	L	256	SKA	HPC3	7 647	n/s*	102 400	5 121	n/s*
	A	1	sRCA	HPC3	172	142	220	4 522 241	9 725
	R	1	sRCA	HPC2	178	148	110	6 553 857	12 319
	ALRP	1	pRCA	HPC2	192	178	158	81 921	144
	ALP	8	SKA	HPC3	391	344	3 200	13 057	23
2	L	256	SKA	HPC3	17 621	n/s*	307 200	5 121	n/s*
	A	1	sRCA	HPC3	386	193	660	4 522 241	8 004
	R	1	sRCA	HPC2	403	204	330	6 553 857	11 600
	ALRP	1	pRCA	HPC2	435	248	474	81 921	144
	ALP	8	SKA	HPC3	893	580	9 600	13 057	25

*not synthesizeable within 72 hours

6.6 Kyber

Kyber is a PQC Key Encapsulation Mechanism (KEM), and will be standardized by NIST under the name *ML-KEM*. Still, to the best of our knowledge, no fully masked – and thus, comprehensively secured against side-channel attacks – implementation has yet been published. For details about our implementation, we refer to Appendix C.

For KEMs in general, the decapsulation is the most critical operation from a side-channel point of view. Since Kyber uses the Fujisaki-Okamoto transform, decapsulation consists of a Chosen Plaintext Attack (CPA)-secure decryption and a deterministic re-encryption. In fact, the re-encryption is particularly security-critical [ABH+22], and thus, it must be weighed for each specific use case whether a CPA decryption is sufficient or the Chosen Ciphertext Attack (CCA)-secure decapsulation is required. Indeed, Düzlü et *al.* recently presented a lightweight identification protocol [DKPS23], which requires side-channel security only for the CPA decryption.

Consequently, we provide implementations for both cases. Table 6 shows the DSE and synthesis results for CPA decryption. Our template for decryption features a fully configurable polynomial multiplication and the number of parallel compression steps. Algorithmically, the template can be instantiated for each parameter set separately, but also such that all three parameter sets are supported simultaneously.

For CCA decapsulation, Table 7 shows our DSE and synthesis results. Remarkably, our template structure for Kyber-CCA yields more than *one million* valid design choices, each of which we have explored.

Table 6. DSE and synthesis results for Kyber-512-CPA decryption. Results on the other parameter sets and a design with runtime-configurable security level can be found in Appendix D (Table 16).

d	Opt.	**Design Config.**				**Area**		**Rand.**	**Latency**	**Delay**
		#comp.	#add.	adder	gadget	[e. kGE]	[kGE]	[bit]	[cycles]	[ms]
0	L	256	256	SKA	—	454	n/s*	—	3 333	n/s*
	A	1	1	sRCA	—	42	120	—	4 982 148	9.03
	ALP	20	32	SKA	—	93	247	—	6 929	0.02
1	L	256	256	SKA	HPC3	7 737	n/s*	104 448	11 016	n/s*
	A	1	1	sRCA	HPC3	179	175	228	9 045 511	18.73
	R	1	1	sRCA	HPC2	186	180	114	13 108 746	25.45
	ALRP	1	1	pRCA	HPC2	200	211	162	164 874	0.29
	ALP	4	8	SKA	HPC3	399	380	3 232	26 951	0.05
2	L	256	256	SKA	HPC3	17 834	n/s*	313 344	11 016	n/s*
	A	1	1	sRCA	HPC3	403	233	684	9 045 511	20.79
	R	1	1	sRCA	HPC2	419	244	342	13 108 746	28.68
	ALRP	1	1	pRCA	HPC2	451	289	486	164 874	0.29
	ALP	3	8	SKA	HPC3	910	642	9 672	26 973	0.06

*not synthesizeable within 72 hours

Table 7. DSE and synthesis results for Kyber-512-CCA decapsulation. Results for all parameter sets that also include the design configurations can be found in Appendix D (Table 17).

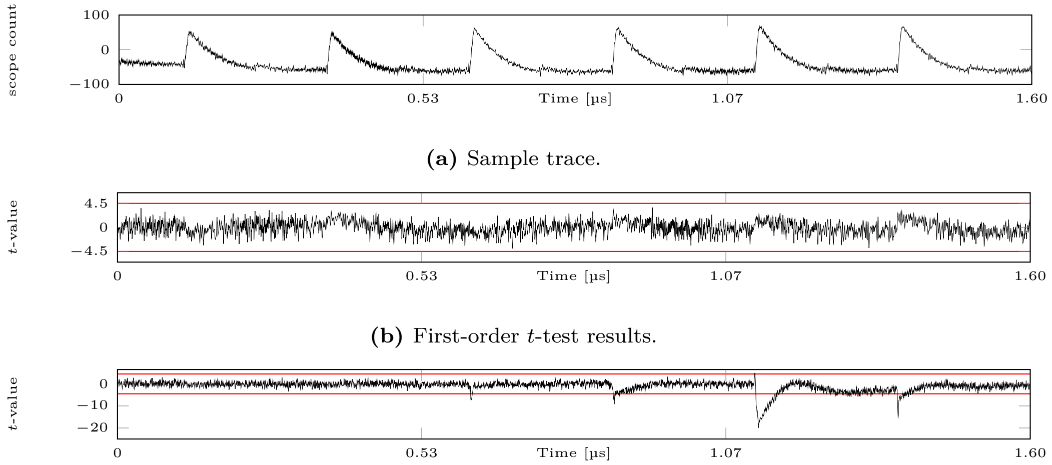
d	Opt.	**Area**		**Rand.**	**Latency**	**Delay**	**SRAM**
		[est. kGE]	[kGE]	[bit]	[cycles]	[μs]	[bit]
0	L	490	n/s*	—	19 612	n/s*	26 816
	A	75	316	—	19 960 068	75 321	
	ALP	91	459	—	72 036	541	
1	L	7 869	n/s*	105 922	69 220	n/s*	28 608
	A	259	447	326	36 368 922	136 213	
	R	268	454	163	52 707 120	205 887	
	ALRP	282	485	211	931 632	3 741	
	ALP	505	668	3 880	138 441	541	
2	L	18 095	n/s*	317 766	69 220	n/s*	30 400
	A	544	575	978	36 368 922	137 241	
	R	564	590	489	52 707 120	198 895	
	ALRP	596	635	633	931 632	3 516	
	ALP	1 113	1 013	11 640	138 441	543	

*not synthesizeable within 72 hours

7 Evaluation

In order to demonstrate that the masked designs generated by our proof-of-concept implementation of HADES are secure against side-channel attacks, we exemplary perform a Test Vector Leakage Assessment (TVLA) on a 16-bit adder. More precisely, we configure our tool to generate a first-order secure 16-bit Sklansky adder using HPC3 gadgets. We synthesize the resulting Verilog file for the side-channel measurements board Sakura-G which is equipped with a Spartan 6 Field-Programmable Gate Array (FPGA). To avoid any recombination of shares in the FPGA fabric, we reimplemented the first-order HPC3 gadget directly using Xilinx primitives, i.e., Look-Up Tables (LUTs) and FDRE registers. The measurement board is connected to a ZFL-2000GH+ Low Noise Amplifier (LNA) configured with a gain of 22.5 dB and a Spectrum M4 oscilloscope (8 bit resolution). The oscilloscope is configured to sample the power consumption with 2.5 GS/s. To provide the required randomness for the underlying HPC3 gadgets of the adder, we instantiate a Keccak core with a state width of 200 bits serving as Pseudorandom Number Generator (PRNG).

For the TVLA, we apply Welch's t-test to investigate if possible leakage can be detected within the power traces. Commonly, a threshold of ±4.5 is used to decide whether leakage can be observed or not. To this end, Figure 3 presents the corresponding measurement results. For reference, Figure 3a shows a sample trace of the addition performed by the protected Sklansky adder. Figure 3b presents the t-test results based on the first-statistical moment using 100 million power traces. As shown, no value exceeds the threshold which means that variations in the mean do not leak information about the processed data. Since the measured design is only protected against first-order attacks, the evaluation based on the second-statistical moment shows some leakage (see Figure 3c).



(a) Sample trace.

(b) First-order t-test results.

(c) Second-order t-test results.

Figure 3. Measurement results for a 16-bit Sklansky adder protected by HPC3 gadgets for $d = 1$ (100 Million traces).

8 Discussion and Comparison

This section briefly discusses the performance, capabilities, and limitations of our proof-of-concept implementation.

8.1 DSE Performance

All experiments, exhaustively searching the design spaces, were performed on a virtual machine running Ubuntu 22.04 with 32 CPUs and 128 GB of RAM. Table 8 shows the number of configurations and the DSE runtime for all case studies in Section 6.

As expected, the number of configurations has a significant influence on the runtime of the DSE. However, considering the large difference between the runtimes of Kyber-CPA and Kyber-CCA, the higher complexity of the template hierarchy for Kyber-CCA clearly influences the runtime as well.

Table 8. Number of configurations and DSE runtime for the algorithms from our case studies.

Algorithm	# Configurations	Time
Keccak	14	0.5s
AdderModQ	42	0.7s
Sparse Polynomial Multiplication	372	1.2s
ChaCha20	1 080	3.2s
AES	1 440	5.4s
Polynomial Multiplication	1 302	7.9s
Kyber-CPA	40 362	196.5s
Kyber-CCA	1 148 364	>150h

Table 9. Comparison to AGEMA, all designs synthesized with the NanGate 45 nm library under the same synthesis options. The AGEMA results use the same hardware description as a starting point.

d	Gadget	Tool	Ripple-Carry		Kogge-Stone		Sklansky	
			Area	Delay	Area	Delay	Area	Delay
			[kGE]	[ns]	[kGE]	[ns]	[kGE]	[ns]
0	—	HADES	1.2	33.7	0.7	0.55	0.5	0.68
1	HPC2	AGEMA	54.3	315.2	35.1	4.46	25.6	5.64
		HADES	2.5	107.8	26.5	3.36	16.6	5.28
	HPC3	AGEMA	37.3	210.8	23.0	2.46	16.2	3.30
		HADES	2.4	73.9	18.7	1.86	11.7	2.64
2	HPC2	AGEMA	126.8	241.7	77.0	5.28	53.4	6.72
		HADES	3.8	140.1	64.1	3.96	40.0	6.84
	HPC3	AGEMA	89.9	153.0	51.1	2.82	34.7	3.60
		HADES	3.6	82.5	44.7	2.22	27.9	3.48

8.2 Comparison

Due to the novelty of our approach, we are not aware of any other works that have a similar concept and would thus be suited for a comparison. In the realm of design automation for masked hardware, there is only AGEMA [KSM20], which is a pure netlist transformer, allowing to automatically mask netlists of unprotected circuits. As AGEMA relies on already synthesized designs, it does not enable DSE as HADES does. Essentially, AGEMA works on a lower layer and only provides a subset of our functionality.

Nevertheless, a fair comparison is possible as both tools produce masked designs. For the sake of simplicity, Table 9 compares selected synthesis and performance results for masked addition units generated both with AGEMA and our tool. However, we assume that the results also scale for more complex modules.

As a baseline, we synthesized three different non-protected addition modules, namely a pipelined KSA and SKA, and a serial RCA, generated from our three addition templates. Each adder has a width of 32 bit, and the Boolean gates were configured to have no additional register stage. Using AGEMA, the synthesized netlists are then masked at first and second order using both the HPC2 and HPC3 PINI-gadgets. Similarly, we generated first-order and second-order masked instances for all three addition templates with the same set of PINI-gadgets using our tool. For performance comparison, all masked designs were synthesized again, providing accurate figures for area occupation and critical path delay (cf. Table 9). Notably, across all adder types, security orders, and gadget types, the results generated by our tool predominantly outperform the AGEMA-generated designs in the acquired design metrics.

This clearly emphasizes the limitations of post-synthesis transformation (AGEMA) compared to pre-synthesis instantiation (our tool) of gadget-based masking schemes. The outcome of AGEMA is heavily dependent on the provided synthesized gate-level netlists, i.e., the processing and optimization procedures of the employed synthesizer, while our tool is directly operating on the pre-synthesis hardware model. Combined with a DSE, our tool plays to its full strengths, as it rapidly configures and compares different designs to find the best candidates, while AGEMA requires manual interaction and time-consuming design iterations to improve efficiency. However, these benefits come at the expense of reduced versatility, since custom template-based hardware model definitions are required, while AGEMA is model-agnostic and handles arbitrary synthesized gate-level netlists.

For a comparison with related work regarding single case studies, we refer to the next subsection (AES), Appendix E (Keccak), and Appendix F (Kyber).

8.3 Limitations and Future Work

Performance Metric Selection

As visible in Section 6, the SKA is constantly picked over the KSA as the configuration for adders during DSE. Both adders yield the same cycle count latency, and the SKA has a lower area estimate as a tie breaker, so this choice is reasonable for latency optimization. However, synthesizing both adder types – KSA results are shown in Table 10 – leads to a converse result regarding the delay: the KSA has a slightly but noteworthy lower delay than the SKA because of the lower fan-out. This discrepancy between the DSE results and the actual performance highlights that if the applied PMs reflect the desired model characteristics insufficiently, they cannot be optimized during DSE.

Our case studies also highlight that the area estimation differs from the actual post-synthesis area with varying margins of error. An underestimation, such as in most designs in Table 7, occurs if the template has a complex FSM or large multiplexers, as these are not entirely captured by the performance prediction. On the other hand, overestimations of the area happen if the synthesis tool is able to optimize the design in terms of area (cf. Table 6). Improving the accuracy of the area estimation would certainly be possible with more sophisticated methods that capture FSMs and Multiplexers or consider possible optimizations, however at the prize of less efficient performance prediction.

Table 10. Kogge-Stone Adder synthesis results for 16-bit addition, cf. Table 1.

d	Gadget	Area		Rand.	Latency	Delay
		[est. kGE]	[kGE]	[bit]	[cycles]	[ns]
0	—	0.277	0.277	—	0	0.45
1	HPC2	16.5	13.4	92	10	2.8
	HPC3	10.2	8.7	184	5	1.6
2	HPC2	38.7	28.9	276	10	3.5
	HPC3	23.9	19.0	552	5	1.9

Model Optimization

Table 11 exemplarily shows a comparison with a wide selection [MPL+11, GMK16, CRB+16, SM21a, CBR+15] of state-of-the-art AES implementations, including designs that were automatically masked by AGEMA. Strikingly, our gadget-based designs hardly compete with the handcrafted designs, despite minimizing several PMs during DSE. In fact, the handcrafted designs manually incorporate various low-level and algorithmic optimization techniques, e.g., the *changing of the guards* methodology [Dae17] to improve

Table 11. Comparison with previous work for masked AES implementations. Note that our designs and the ones from [KSM20] are the only ones to use gadget-based masking, while all other designs are handcrafted and manually optimized.

Ref.	d	**Area**	**Rand.**	**Latency**	**Delay**	**Technology**
		[kGE]	[bit]	[bit]	[ns]	
[MPL+11]	1	11.1	48	266		UMCL18G212T3
[GMK16]	1	7.6	28	216		UMC 180 nm
[CRB+16]	1	6.7	54	276		NanGate 45 nm
[SM21a]	1	7.1	1	246	1 537	UMC 180 nm
[KSM20]	1	33.1	414	3 859	9 879	NanGate 45 nm
[KSM20]	1	10.0	34	2 043	4 310	NanGate 45 nm
[KSM20]	1	52.6	680	99	202	NanGate 45 nm
this	1	23.8	144	1 616	1 972	NanGate 45 nm
this	1	26.1	68	2 576	3 503	NanGate 45 nm
this	1	860.8	11 592	51	83	NanGate 45 nm
this	1	125.9	1 224	55	100	NanGate 45 nm
[CBR+15]	2	18.6	126	276		NanGate 45 nm
[GMK16]	2	12.8	84	216		UMC 180 nm
[KSM20]	2	17.6	102	2 043	5 434	NanGate 45 nm
[KSM20]	2	131.6	2 040	99	237	NanGate 45 nm
this	2	40.0	408	1 616	2 101	NanGate 45 nm
this	2	45.2	204	2 576	3 812	NanGate 45 nm
this	2	1 670.8	34 764	51	115	NanGate 45 nm
this	2	227.0	3 660	55	116	NanGate 45 nm

efficiency and performance. This, however, is not yet reflected by our tool and the DSE which solely focuses on *generation* and *exploration* of hardware designs without further consideration of design *optimization*.

Still, we would like to emphasize that this limitation only affects our proof-of-concept implementation of HADES, not the general concept as discussed in Section 3. For this, future versions of our tool could be enhanced, e.g., for randomness reduction [FKS+22, KM22a] or masking conversion, to boost performance and efficiency of automatically generated hardware instances even further.

Nevertheless, it is remarkable that our automatically generated designs achieve similar or better performances compared to the ones from AGEMA. Also, it is important to note that generating the wide variety of our AES designs took about *five seconds* (cf. Table 8), whereas all other implementations featured in the table do not offer this flexibility at all.

9 Conclusion

In this work, we present a novel concept of describing cryptographic hardware, which is incorporated by our versatile HADES framework. The accompanying template library covers a wide range of use cases and is easily extensible, re-usable, and will be available publicly. Most importantly, HADES enables designers to perform DSE rapidly and based on objective PMs, rather than relying on intuition, experience, or trial-and-error.

Notably, our use case study yielded pioneering and competitive ASIC implementation results for many algorithms. In particular, to the best of our knowledge, we present the first set of masked hardware implementations of ChaCha20, and the first arbitrary-order masked hardware modules for the designated PQC standard Kyber.

Acknowledgements

The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and 510964147 (CAVE). Moreover, the authors have been funded by the European Commission under the grant agreement number 101070374 (CONVOLVE), and by the German Federal Ministry of Education and Research (BMBF) through the projects VE-HEP (16KIS1345) and 6GEM (16KISK038).

References

[ABC$^+$23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR TCHES*, 2023(4):58–79, 2023.

[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.

[ABH$^+$22] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. Systematic Study of Decryption and Re-encryption Leakage: The Case of Kyber. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*, volume 13211 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 2022.

[BDN$^+$13] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013.

[Ber19] D. J. Bernstein. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. online, 2019. https://competitions.cr.yp.to/caesar.html.

[BG22] Florian Bache and Tim Güneysu. Boolean Masking for Arithmetic Additions at Arbitrary Order in Hardware. *Applied Sciences*, 12(5), 2022.

[BGR$^+$21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. *IACR TCHES*, 2021(4):173–214, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064.

[BNG21] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. High-performance hardware implementation of crystals-dilithium. In *FPT*, pages 1–10. IEEE, 2021.

[BSG23] Fabian Buschkowski, Pascal Sasdrich, and Tim Güneysu. EASIMask - Towards Efficient, Automated, and Secure Implementation of Masking in Hardware. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[CBR+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES s-box. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2015.

[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.

[Chi23] Constructing Hardware in a Scala Embedded Language (Chisel). online, 2023.

[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, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[CRB+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS 2016 Vienna, Austria, October, 2016*, page 43. ACM, 2016.

[Dae17] Joan Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 137–153. Springer, Heidelberg, September 2017.

[DKPS23] Samed Düzlü, Juliane Krämer, Thomas Pöppelmann, and Patrick Struck. A lightweight identification protocol based on lattices. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 95–113. Springer, Heidelberg, May 2023.

[FKS+22] Jakob Feldtkeller, David Knichel, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Randomness optimization for gadget compositions in higher-order masking. *IACR TCHES*, 2022(4):188–227, 2022.

[Gaj] Kris Gaj. ATHENa: Automated Tools for Hardware EvaluatioN. online. https://cryptography.gmu.edu/athena/.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-Order Side-Channel Protected Implementations of KECCAK. In *DSD*, pages 205–212. IEEE Computer Society, 2017.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

[KLRBG23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A holistic approach towards side-channel secure fixed-weight polynomial sampling. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part II*, volume 13941 of *LNCS*, pages 94–124. Springer, Heidelberg, May 2023.

[KM22a] David Knichel and Amir Moradi. Composable gadgets with reused fresh masks first-order probing-secure hardware circuits with only 6 fresh masks. *IACR TCHES*, 2022(3):114–140, 2022.

[KM22b] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1799–1812. ACM Press, November 2022.

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR TCHES*, 2022(1):589–629, 2022.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 787–816. Springer, Heidelberg, December 2020.

[LMRG24] Georg Land, Adrian Marotzke, Jan Richter-Brockmann, and Tim Güneysu. Gadget-based masking of streamlined NTRU prime decapsulation in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):1–26, 2024.

[LSG21] Georg Land, Pascal Sasdrich, and Tim Güneysu. A hard crystal - implementing dilithium on reconfigurable hardware. In *CARDIS*, volume 13173 of *Lecture Notes in Computer Science*, pages 210–230. Springer, 2021.

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.

[NIS17] NIST. Call for Proposals – Post-Quantum Cryptography. Technical Report, CSRC, 2017. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals.

[RBCGG22] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved polynomial multiplication and inversion in hardware. *IACR TCHES*, 2022(1):557–588, 2022.

[RMG22] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding BIKE: scalable hardware implementation for reconfigurable devices. *IEEE Trans. Computers*, 71(5):1204–1215, 2022.

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

[SM21a] Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes nullifying fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):305–342, 2021.

[SM21b] Aein Rezaei Shahmirzadi and Amir Moradi. Second-order SCA security with almost no fresh randomness. *IACR TCHES*, 2021(3):708–755, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8990.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking - towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 559–578. Springer, Heidelberg, June 2015.

[Spi23] SpinalHDL. online, 2023.

[ZZW$^+$22] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for CRYSTALS-dilithium. *IACR TCHES*, 2022(1):270–295, 2022.

A Templates for Modular Addition

Several contemporary use cases, such as Kyber, require addition in $\mathbb{Z}_q$ for non-power-of-two q. For this, we implement a template that has freely configurable q. As proposed in [LMRG24], the general idea is to add the inputs a and b and then subtract q while storing the original addition result. As a result, we have $a + b$ and $a + b - q$. Finally, we can use the carry-out of the subtraction to select between both results.

The modular addition template employs the basic adder templates, automatically choosing the correct bit widths. Consequently, this template can be instantiated as a serial or pipelined module as well, depending on which basic adder is utilized. Table 12 shows the DSE and synthesis results for the moduli of Kyber and Dilithium. Interestingly, for the larger q, the serial RCA is chosen for $d = 0$ with area optimization. This can be explained by the fact that for the big modulus, the size of generate/propagate groups outweighs the size of the register stage.

Table 12. DSE and synthesis results for modular addition. Exemplarily, we choose the moduli of Kyber and Dilithium, but the template allows parametrization of any modulus.

d	**Opt.**	**Design Config.**		**Area**		**Rand.**	**Lat.**	**Delay**
		gadget	adder	[est. kGE]	[kGE]	[bits]	[cycles]	[ns]
Algorithm Parameter: $q = 3329$ (Kyber)								
0	L/A/ALP	—	SKA	0.4	0.4	—	0	1.0
1	L	HPC3	SKA	15.8	16.1	208	11	4.7
	A/ALRP	HPC3	sRCA	2.8	2.6	28	61	63.4
	R/ALP	HPC2	sRCA	3.5	3.2	14	88	81.0
2	L	HPC3	SKA	36.4	34.5	966	11	5.6
	A/ALRP	HPC3	sRCA	6.3	4.7	84	61	60.4
	R/ALP	HPC2	sRCA	8.0	6.1	42	88	106.5
Algorithm Parameter: $q = 8380417$ (Dilithium)								
0	L/ALP	—	SKA	0.8	0.8	—	0	1.2
	A	—	sRCA	0.8	1.4	—	56	54.3
1	L/ALP	HPC3	SKA	36.1	38.2	478	13	7.5
	A	HPC3	sRCA	4.9	4.7	50	105	105.0
	R/ALRP	HPC2	sRCA	6.2	5.8	25	154	163.3
2	L/ALP	HPC3	SKA	83.4	82.0	1 434	13	6.8
	A	HPC3	sRCA	11.0	8.5	150	105	115.5
	R/ALRP	HPC2	sRCA	14.2	10.9	75	154	201.8

B Dilithium Multiplication Implementation Details

Recall from Section 6.5 that the Dilithium sparse multiplication (between the secret key and the challenge polynomial c) is an operation that is highly vulnerable to side-channel attacks. This multiplication can be carried out by accumulating the secret key coefficients rotated by each public offset. Specifically for Dilithium, each offset has a sign associated since the challenge is ternary (i.e., the non-zero coefficients are either one or minus one). Since every adder can act as a subtracter simply by inverting the subtrahend and adding one as carry-in, this scenario can also be covered easily with our already existing templates. The resulting polynomial has signed integer coefficients which are not reduced modulo q. However, the sparsity τ times the secret key range η (this product represents the maximum possible result coefficient) is smaller than the modulus for each parameter set, so the reduction can be performed together with the subsequent addition to the nonce $\mathbf{y}$.

Remarkably, the multiplications of the secret key polynomials $\mathbf{s}_1$ and $\mathbf{s}_2$ with the challenge are the only polynomial multiplications in Dilithium signing that *must* be performed masked for a side-channel secure implementation. The only other multiplication – $\mathbf{Ay}$ – has a public operand as well, but the secret operand has a wide distribution, making a gadget-based approach infeasible. However, since the nonce $\mathbf{y}$ does not depend on the secret key and is different for each message to be signed, it is likely that protection against simple power analysis is sufficient and thus, masking may be optional. By all means, this operation is less critical than the multiplication that involves the secret key. Overall, a side-channel secure implementation of Dilithium only based on secure gadgets may be feasible, but we leave this as future work.

Table 13 features DSE and synthesis results.

Table 13. Sparse polynomial multiplication for the Dilithium use case with different parameters for η (the range of the secret input coefficients) and τ (the sparsity of the public polynomial).

d	**Opt.**	**Design Cfg.**			**Area**		**Rand.**	**Latency**	**Delay**
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
Algorithm Parameters: sparsity $\tau = 39$, range of secret $\eta = 2$									
0	L	256	SKA	—	24	n/s*	—	79	n/s*
	A	1	SKA	—	1	16	—	10 024	6.8
	ALP	16	SKA	—	5	197	—	664	0.5
1	L	256	SKA	HPC3	1 096	4 601	13 312	274	1.0
	A	1	RCA	HPC3	3	24	36	10 375	16.7
	R/ALRP	1	RCA	HPC2	5	28	18	10 726	17.3
	ALP	2	RCA	HPC3	4	48	72	5 383	10.9
2	L	256	SKA	HPC3	2 522	n/s*	39 936	274	n/s*
	A	1	RCA	HPC3	6	35	108	10 375	19.4
	R/ALRP	1	RCA	HPC2	10	39	54	10 726	20.0
	ALP	2	RCA	HPC3	8	67	216	5 383	10.8
3	L	256	SKA	HPC3	4 535	n/s*	79 872	274	n/s*
	A	1	RCA	HPC3	10	45	216	10 375	19.4
	R/ALRP	1	RCA	HPC2	15	51	108	10 726	20.0
	ALP	2	RCA	HPC3	14	87	432	5 383	10.9
Algorithm Parameters: sparsity $\tau = 49$, range of secret $\eta = 4$									
0	L	256	SKA	—	27	n/s*	—	99	n/s*
	A	1	SKA	—	1	19	—	12 594	8.7
	ALP	16	SKA	—	2	216	—	834	0.7
1	L	256	SKA	HPC3	1 192	n/s*	14 336	344	n/s*
	A	1	RCA	HPC3	4	30	40	13 084	16.7
	R/ALRP	1	RCA	HPC2	6	32	20	13 574	17.3
	ALP	2	RCA	HPC3	4	55	80	6 812	13.6
2	L	256	SKA	HPC3	2 744	n/s*	43 008	344	n/s*
	A	1	RCA	HPC3	7	40	120	13 084	24.3
	R/ALRP	1	RCA	HPC2	11	45	60	13 574	25.2
	ALP	2	RCA	HPC3	9	76	240	6 812	13.8
3	L	256	SKA	HPC3	4 931	n/s*	86 016	344	n/s*
	A	1	RCA	HPC3	11	51	240	13 084	24.3
	R/ALRP	1	RCA	HPC2	18	58	120	13 574	25.2
	ALP	2	RCA	HPC3	16	99	480	6 812	13.8
Algorithm Parameters: sparsity $\tau = 60$, range of secret $\eta = 2$									
0	L	256	SKA	—	24	n/s*	—	121	n/s*
	A	1	SKA	—	1	17	—	15 421	10.5
	ALP	16	SKA	—	2	194	—	1 021	0.7
1	L	256	SKA	HPC3	1 096	n/s*	13 312	421	n/s*
	A	1	RCA	HPC3	3	26	36	15 961	25.7
	R/ALRP	1	RCA	HPC2	5	28	18	16 501	26.6
	ALP	2	RCA	HPC3	4	49	72	8 281	16.7
2	L	256	SKA	HPC3	2 522	n/s*	39 936	421	n/s*
	A	1	RCA	HPC3	6	36	108	15 961	29.7
	R/ALRP	1	RCA	HPC2	10	39	54	16 501	30.7
	ALP	2	RCA	HPC3	8	68	216	8 281	16.7
3	L	256	SKA	HPC3	4 534	n/s*	79 872	421	n/s*
	A	1	RCA	HPC3	10	45	216	15 961	29.7
	R/ALRP	1	RCA	HPC2	15	51	108	16 501	30.7
	ALP	2	RCA	HPC3	14	88	432	8 281	16.8

*not synthesizeable within 72 hours

C Kyber Implementation Details

Kyber operates on the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256}+1)$ with $q = 3329$ for all parameter sets. All notation is borrowed from the official Kyber specification [ABD+21]. The central assumption under which this approach to implementing Kyber works is that the secret key is not stored in Number-Theoretic Transform (NTT) representation, as we want to avoid NTT operations on secrets. Importantly, this is reasonable as the Kyber key generation can be performed using the very same secure multiplication that we deploy for decryption and then storing the secret key Boolean-masked. Positively, this reduces the memory footprint of the secret key by a factor of 4.

In the following, we detail about how to implement Kyber using secure Boolean gadgets only. Generally, we follow the recent idea of using gadget-based masking for polynomial multiplication, which has been introduced and shown practical for Streamlined NTRU Prime in [LMRG24]. In this paper, the authors also discuss a potential applicability to Kyber, yet leaving out many details.

CPA decryption

In our implementation, we start by reading the ciphertext polynomial v into the result register of the polynomial multiplication module. During reading, we additionally

- decompose each coefficient (which is a multiplication by q where only some upper result bits are computed),
- negate the decomposition result, and
- subtract $\lceil q/4 \rfloor$.

This procedure is employed to prepare for the compression of the final result.

Subsequently, we perform the vector-vector multiplication of $\mathbf{s}$ and $\mathbf{u}$, accumulating this result to the preprocessed v. Thus, we compute $\mathbf{su} + (-v - \sum_{i=0}^{255} \lceil q/4 \rfloor X^i)$, which deviates from the Kyber specification. However, it enables omitting an explicit negation of the secret vector-vector multiplication result. Notably, the Kyber compression is symmetric and directly works on this inverted result. The constant offset is subtracted due to the compression function presented by Bos et *al.* [BGR+21, Alg. 1], which we adapt to the fully Boolean-masked setting.

CCA decapsulation

The decapsulation is considerably more complex as depicted in Algorithm 2. In addition to the modules required by the decryption, it requires

- a side-channel secure Keccak for different SHA3 and SHAKE variations,
- side-channel secure Centered Binomial Distribution (CBD) sampling,
- an inverse NTT module for *public* data, and
- a Keccak module for expanding the matrix $\hat{\mathbf{A}}$.

In principle, expanding $\hat{\mathbf{A}}$ with the side-channel secure Keccak module would also be possible, but this would induce an unnecessary delay since $\hat{\mathbf{A}}$ is public. Moreover, the polynomials in $\hat{\mathbf{A}}$ are always generated when the side-channel secure Keccak generates secret polynomials in parallel. Besides, the inverse NTT is required for compliance with the Kyber specification, where for performance reasons, $\hat{\mathbf{A}}$ is generated in NTT domain. On the contrary, we assume $\mathbf{t}$ to be in non-NTT domain because a key generation module

Algorithm 2 Kyber-CCA decapsulation without NTT multiplication

```
1: procedure DECAPS(ciphertext $(\overline{v}, \overline{\mathbf{u}})$, secret key $(\mathbf{s}, \rho, \mathbf{t}, h, z) \in \mathcal{R}_q^k \times \mathcal{B}^{32} \times \mathcal{R}_q^k \times \mathcal{B}^{32} \times \mathcal{B}^{32}$)
2:     decompress $(\overline{v}, \overline{\mathbf{u}})$ to $(v, \mathbf{u}) \in \mathcal{R}_q \times \mathcal{R}_q^k$
3:     $m \in \mathcal{B}^{32} := \mathsf{Decrypt}_{\mathsf{CPA}}(v, \mathbf{u}, \mathbf{s})$
4:     $(K', coins) \in \mathcal{B}^{32} \times \mathcal{B}^{32} := \mathsf{SHA3\text{-}512}(m||h)$
5:     expand $\hat{\mathbf{A}}$ from $\rho$
6:     $\mathbf{A} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}})$
7:     sample $\mathbf{r}, \mathbf{e_1}, e_2$ deterministically from $coins$
8:     $\mathbf{u}' := \mathbf{A}^T\mathbf{r} + \mathbf{e_1}$
9:     $v' := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}(m, 1)$
10:    if $(v', \mathbf{u}') \approx (v, \mathbf{u})$ [BGR+21, Alg. 2] then
11:        $K := \mathsf{SHAKE\text{-}256}(K'||\mathsf{SHA3\text{-}256}(c))$
12:    else
13:        $K := \mathsf{SHAKE\text{-}256}(z||\mathsf{SHA3\text{-}256}(c))$
14:    return $K$
```

that employs gadget-based masking with Schoolbook multiplication would produce $\mathbf{t}$ that way.

As indicated in Line 10 of Algorithm 2, we use the *decompressed comparison* technique by Bos et *al.* [BGR+21, Alg. 2] to perform the comparison between the received ciphertext and the deterministic re-encryption. Adapting this method to a fully Boolean masking setting, we instantiate an additional modular addition at the top level[1], which is used for the modular additions with the range constants that depend on the ciphertext coefficients. These range constants are also the main source of SRAM usage for our designs, as there are 1040 12-bit values for Kyber-512 and -768, and 2080 such values for Kyber-1024.

Moreover, we require an additional 12-bit adder module for a post-processing step after CBD sampling. This sampling procedure is defined such that signed three-bit coefficients are the results. For further computations, however, we require them to have the non-negative modular representation. Consequently, we conditionally add q based on the sign bit of the CBD sample.

[1] A more sophisticated template might be able to avoid this by re-using an adder from the polynomial multiplication.

D Additional Results

Here, we include several other results from our case studies that we left out in the main body, namely:

- full AES DSE and synthesis results (Table 14)
- additional parametrizations and third order masking syntheses for polynomial multiplication (Table 15)
- synthesis results for Kyber-768 and -1024 CPA decryption and a design that enables runtime configurability regarding the parameter set (Table 16)
- full synthesis results for Kyber CCA decapsulation (Table 17)

In Table 17, we omit the SRAM usage for space reasons. Kyber-512 requires 26 816 bit, 28 608 bit, and 30 400 bit for $d \in \{0, 1, 2\}$. Kyber-768 requires 33 216 bit, 35 776 bit, and 38 336 bit for $d \in \{0, 1, 2\}$. Kyber-1024 requires 53 376 bit, 56 704 bit, and 60 032 bit for $d \in \{0, 1, 2\}$.

Table 14. DSE and synthesis results for AES

d	**Opt.**	**Design Configuration**						**Area**		**Rand.**	**Latency**	**Delay**
		architecture	rSBox	#rSBox	#MC	kSBox	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[ns]
				Algorithm Parameter:	AES-128							
0	L/ALP	round	Can	16	4	Can	—	17.3	32.9	—	15	501
	A	round	Can	1	1	Can	—	4.4	9.9	—	656	1 292
1	L	unrolled	Can	16	1	Can	HPC3	814.4	860.8	11 592	51	83
	A	round	Can	1	1	Can	HPC3	20.0	23.8	144	1 616	1 972
	R/ALRP	round	BP	1	1	BP	HPC2	22.1	26.1	68	2 576	3 503
	ALP	round	Can	16	1	Can	HPC3	119.5	125.9	1 224	55	100
2	L	unrolled	Can	16	4	Can	HPC3	1 819.3	1 670.8	34 764	51	115
	A	round	BP	1	1	BP	HPC3	44.1	40.0	408	1 616	2 101
	R/ALRP	round	BP	1	1	BP	HPC2	49.2	45.2	204	2 576	3 812
	ALP	round	Can	16	4	Can	HPC3	267.3	227.0	3 660	55	116
				Algorithm Parameter:	AES-192							
0	L/ALP	round	Can	16	4	Can	—	19.5	38.0	—	17	585
	A	round	Can	1	1	Can	—	5.1	12.2	—	778	1 533
1	L	unrolled	Can	16	1	Can	HPC3	977.6	1 034.9	13 896	61	111
	A	round	Can	1	1	Can	HPC3	22.9	28.6	144	1 866	2 836
	R/ALRP	round	BP	1	1	BP	HPC2	25.0	30.8	68	2 954	4 549
	ALP	round	Can	16	1	Can	HPC3	128.2	135.9	1 224	65	118
2	L	unrolled	Can	16	1	BP	HPC3	2 183.3	1 992.4	41 676	61	116
	A	round	BP	1	1	BP	HPC3	50.8	47.1	408	1 866	2 724
	R/ALRP	round	BP	1	1	BP	HPC2	55.8	52.2	204	2 954	5 849
	ALP	round	Can	16	4	Can	HPC3	286.9	242.4	3 660	65	124
				Algorithm Parameter:	AES-256							
0	L/ALP	round	Can	16	4	Can	—	21.3	41.4	—	19	636
	A	round	Can	1	1	Can	—	5.5	12.9	—	1 378	2 715
1	L	unrolled	Can	16	1	Can	HPC3	1 139.2	1 205.3	16 200	71	133
	A	round	Can	1	1	Can	HPC3	24.4	29.9	144	2 946	3 948
	R/ALRP	round	BP	1	1	BP	HPC2	26.5	32.2	68	4 514	6 274
	ALP	round	Can	16	4	Can	HPC3	135.5	142.8	1 224	75	156
2	L	unrolled	Can	16	1	BP	HPC3	2 545.1	2 321.1	48 588	71	135
	A	round	BP	1	1	BP	HPC3	53.9	49.1	408	2 946	4 419
	R/ALRP	round	BP	1	1	BP	HPC2	62.2	58.2	204	4 514	6 771
	ALP	round	Can	16	1	Can	HPC3	303.2	252.7	3 660	75	143

Table 15. Additional results for polynomial multiplication, cf. Table 5.

d	**Opt.**	**Design Cfg.**			**Area**		**Rand.**	**Latency**	**Delay**
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
Algorithm Parameters: red. poly. $X^{256}+1$, modulus $q = 3329$ range of secret $\eta = 3$ (Kyber-512)									
3	L	256	SKA	HPC3	31 697	n/s*	614 400	5 121	n/s*
	A	1	sRCA	HPC3	688	248	1 320	4 522 241	8 004
	R	1	sRCA	HPC2	717	266	660	6 553 857	11 600
	ALRP	1	pRCA	HPC2	774	329	948	81 921	144.2
	ALP	8	SKA	HPC3	1 599	884	19 200	13 057	22.99
Algorithm Parameters: red. poly. $X^{256}+1$, modulus $q = 3329$ range of secret $\eta = 2$ (Kyber-768 and -1024)									
0	L	256	SKA	—	303	n/s*	—	1 025	n/s*
	A	1	sRCA	—	40	94	—	2 425 089	3 251
	ALP	64	SKA	—	105	186	—	1 793	5
1	L	256	SKA	HPC3	6 111	n/s*	83 968	4 609	n/s*
	A	1	sRCA	HPC3	167	138	148	4 391 169	7 772
	R	1	sRCA	HPC2	171	141	74	6 357 249	8 904
	ALRP	1	pRCA	HPC2	184	172	122	81 153	135
	ALP	10	SKA	HPC3	390	482	3 280	11 009	22
2	L	256	SKA	HPC3	14 100	n/s*	251 904	4 609	n/s*
	A	1	sRCA	HPC3	376	184	444	4 391 169	6 583
	R	1	sRCA	HPC2	385	191	222	6 357 249	9 603
	ALRP	1	pRCA	HPC2	414	236	366	81 153	106
	ALP	10	SKA	HPC3	890	759	9 840	11 009	23
3	L	256	SKA	HPC3	25 381	n/s*	503 808	4 609	n/s*
	A	1	sRCA	HPC3	668	234	888	4 391 169	6 633
	R	1	sRCA	HPC2	685	246	444	6 357 249	9 918
	ALRP	1	pRCA	HPC2	738	308	732	81 153	107.9
	ALP	10	SKA	HPC3	1 595	1 108	19 680	11 009	23.03
Algorithm Parameters: red. poly. $X^{761}-x-1$, modulus $q = 4591$, range of secret $\eta = 1$ (sNTRUp-761)									
0	L	761	SKA	—	530	n/s*	—	2 284	n/s*
	A	1	RCA	—	122	144	—	480 644	721
	ALP	254	SKA	—	257	n/s*	—	3 806	n/s*
1	L	761	SKA	HPC3	18 748	n/s*	307 444	12 177	n/s*
	A	1	sRCA	HPC3	490	208	82	39 960 111	60 363
	R	1	sRCA	HPC2	492	211	41	57 912 862	84 916
	ALRP	1	pRCA	HPC2	506	454	93	626 305	1 040
	ALP	35	SKA	HPC3	1 194	1 306	9 870	28 158	55
2	L	256	SKA	HPC3	43 601	n/s*	922 332	12 177	n/s*
	A	1	sRCA	HPC3	1 101	270	246	39 960 111	62 340
	R	1	sRCA	HPC2	1 106	275	123	57 912 862	87 482
	ALRP	1	pRCA	HPC2	1 139	543	279	626 305	821
	ALP	32	SKA	HPC3	2 588	1 890	27 072	29 680	61
3	L	256	SKA	HPC3	78 774	n/s*	1 844 644	12 177	n/s*
	A	1	sRCA	HPC3	1 958	332	492	39 960 111	60 362
	R	1	sRCA	HPC2	1 968	339	246	57 912 862	90 348
	ALRP	1	pRCA	HPC2	2 025	631	558	626 305	833
	ALP	32	SKA	HPC3	4 636	2 468	54 144	29 680	65

*not synthesizeable within 72 hours

Table 16. Additional DSE and synthesis results for Kyber-CPA decryption, cf. Table 6

d	**Opt.**	**Design Cfg.**				**Area**		**Rand.**	**Lat.**	**Delay**
		#comp.	#add.	adder	gadget	[e. kGE]	[kGE]	[bit]	[kcycles]	[ms]
Algorithm Parameter: Kyber-768 ($k = 3, \eta = 2$)										
0	L	256	256	SKA	—	307	n/s*	—	4 102	n/s*
	A	1	1	sRCA	—	42	120	—	7 276 421	11.42
	ALP	16	43	SKA	—	85	1 180	—	7 957	0.05
1	L	256	256	SKA	HPC3	6 201	n/s*	86 016	14 857	n/s*
	A	1	1	sRCA	HPC3	174	171	156	13 174 792	21.89
	R	1	1	sRCA	HPC2	178	174	78	19 073 035	34.12
	ALRP	1	1	pRCA	HPC2	191	205	126	244 747	0.34
	ALP	3	10	SKA	HPC3	398	518	3 304	34 142	0.07
2	L	256	256	SKA	HPC3	14 314	n/s*	258 048	14 857	n/s*
	A	1	1	sRCA	HPC3	392	225	468	13 174 792	25.68
	R	1	1	sRCA	HPC2	401	232	234	19 073 035	36.82
	ALRP	1	1	pRCA	HPC2	431	277	378	244 747	0.44
	ALP	3	10	SKA	HPC3	908	821	9 912	34 142	0.07
Algorithm Parameter: Kyber-1024 ($k = 4, \eta = 2$)										
0	L	256	256	SKA	—	307	n/s*	—	5	n/s*
	A	1	1	sRCA	—	42	120	—	9 702	14.2
	ALP	16	43	SKA	—	85	1 180	—	11	0.1
1	L	256	256	SKA	HPC3	6 201	n/s*	86 016	20	n/s*
	A	1	1	sRCA	HPC3	174	171	156	17 566	31.8
	R	1	1	sRCA	HPC2	178	174	78	25 431	42.5
	ALRP	1	1	pRCA	HPC2	191	205	126	326	0.5
	ALP	3	8	SKA	HPC3	398	360	2 648	45	0.1
2	L	256	256	SKA	HPC3	14 314	n/s*	258 048	20	n/s*
	A	1	1	sRCA	HPC3	392	225	468	17 566	37.9
	R	1	1	sRCA	HPC2	401	232	234	25 431	49.4
	ALRP	1	1	pRCA	HPC2	431	277	378	326	0.5
	ALP	3	10	SKA	HPC3	908	821	9 912	45	0.1
Algorithm Parameter: runtime-configurable ($k \in \{2, 3, 4\}, \eta \in \{2, 3\}$)										
0	L	256	256	SKA	—	454	n/s*	—	6	n/s*
	A	1	1	sRCA	—	43	121	—	9 964	17.6
	ALP	16	32	SKA	—	93	213	—	14	0.1
1	L	256	256	SKA	HPC3	7 738	n/s*	104 448	22	n/s*
	A	1	1	sRCA	HPC3	180	175	228	18 091	32.0
	R	1	1	sRCA	HPC2	186	181	114	26 217	46.4
	ALRP	1	1	pRCA	HPC2	200	212	162	329	0.6
	ALP	2	8	SKA	HPC3	399	381	3 216	54	0.1
2	L	256	256	SKA	HPC3	17 836	n/s*	313 344	22	n/s*
	A	1	1	sRCA	HPC3	404	234	684	18 091	39.8
	R	1	1	sRCA	HPC2	420	245	342	26 217	57.4
	ALRP	1	1	pRCA	HPC2	452	290	486	329	0.6
	ALP	2	8	SKA	HPC3	910	623	9 648	54	0.1

*not synthesizeable within 72 hours

Table 17. DSE and synthesis results for Kyber-CCA decapsulation.

d	**Opt.**	**Design Configuration**						**Area**		**Rand.**	**Latency**	**Delay**
		poly.mul. # adders	poly.mul. adder type	comparison adder type	cond. add q adder type	Keccak # χ	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
Algorithm parameter: Kyber-512 ($k = 2, \eta = 3$)												
0	L	256	SKA	SKA	SKA	1600	—	490	n/s*	—	19 612	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	75	316	—	19 960 068	75 321
	ALP	10	SKA	SKA	SKA	100	—	91	459	—	72 036	541
1	L	256	SKA	SKA	SKA	1600	HPC3	7 869	n/s*	105 922	69 220	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	259	447	326	36 368 922	136 213
	R	1	sRCA	sRCA	sRCA	25	HPC2	268	454	163	52 707 120	205 887
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	282	485	211	931 632	3 741
	ALP	8	SKA	SKA	sRCA	200	HPC3	505	668	3 880	138 441	541
2	L	256	SKA	SKA	SKA	1600	HPC3	18 095	n/s*	317 766	69 220	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	544	575	978	36 368 922	137 241
	R	1	sRCA	sRCA	sRCA	25	HPC2	564	590	489	52 707 120	198 895
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	596	635	633	931 632	3 516
	ALP	8	SKA	SKA	sRCA	200	HPC3	1 113	1 013	11 640	138 441	543
Algorithm parameter: Kyber-768 ($k = 3, \eta = 2$)												
0	L/ALP	256	SKA	SKA	SKA	1600	—	343	n/s*	—	28 170	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	74	311	—	36 416 346	142 251
1	L	256	SKA	SKA	SKA	1600	HPC3	6 333	n/s*	87 486	107 005	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	254	433	250	66 103 166	250 391
	R	1	sRCA	sRCA	sRCA	25	HPC2	260	439	125	95 698 594	359 769
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	273	469	173	1 557 154	6 254
	ALP	10	SKA	SKA	sRCA	200	HPC3	504	797	3 956	209 081	791
2	L	256	SKA	SKA	SKA	1600	HPC3	14 574	n/s*	26 245	107 005	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	532	553	750	66 103 166	265 475
	R	1	sRCA	sRCA	sRCA	25	HPC2	546	564	375	95 698 594	359 769
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	576	609	519	1 557 154	6 155
	ALP	10	SKA	SKA	sRCA	200	HPC3	1 179	1 179	11 868	209 081	795
Algorithm parameter: Kyber-1024 ($k = 4, \eta = 2$)												
0	L/ALP	256	SKA	SKA	SKA	1600	—	343	n/s*	—	42 122	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	74	312	—	58 260 482	218 204
1	L	256	SKA	SKA	SKA	1600	HPC3	6 333	n/s*	87 486	159 596	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	254	433	250	105 694 008	395 858
	R	1	sRCA	sRCA	sRCA	25	HPC2	260	439	125	153 015 406	577 417
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	273	469	173	2 389 102	9 595
	ALP	10	SKA	SKA	sRCA	200	HPC3	504	797	3 956	322 303	1 259
2	L	256	SKA	SKA	SKA	1600	HPC3	14 574	n/s*	262 458	159 596	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	532	553	750	105 694 008	424 474
	R	1	sRCA	sRCA	sRCA	25	HPC2	546	564	375	153 015 406	614 520
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	576	609	519	2 389 102	9 595
	ALP	10	SKA	SKA	sRCA	200	HPC3	1 110	1 179	11 868	322 303	1 207

*not synthesizeable within 72 hours

E Masked Keccak Comparison

A comparison of our Keccak designs with related work [BDN$^+$13, SM21b, GSM17] is shown in Table 18. As also pointed out for AES, our designs can hardly compete with the handcrafted implementations because several low-level optimizations are not yet available with our tool.

Table 18. Comparison with previous work for masked Keccak modules. Note that our design is the first to employ gadget-based masking and an automated generation process, while all other designs are handcrafted and manually optimized. Notably, our results have been generated within *less than a second* of DSE (cf. Table 8).

Ref.	d	**Area**	**Rand.**	**Latency**	**Delay**	**Technology**
		[kGE]	[bit]	[cycles]	[ns]	
[BDN$^+$13]	1	116.6	4	25	42.2	NanGate 45 nm
[BDN$^+$13]	1	39.0	4	1 625	2 561.2	NanGate 45 nm
[GSM17]	1	18.7	0	3 160	3 690.7	UMC 130 µm
[GSM17]	1	22.3	0	1 648	2 028.8	UMC 130 µm
[GSM17]	1	108.0	0	25	28.2	UMC 130 µm
[SM21b]	1	129.3	0	72	92.9	UMC 130 µm
this	1	149.4	3 200	120	40.8	NanGate 45 nm
this	1	30.1	50	3 144	1 792.5	NanGate 45 nm
this	1	34.6	25	4 680	3 744.0	NanGate 45 nm
this	1	61.6	800	264	198.1	NanGate 45 nm
[GSM17]	2	28.8	75	3 160	3 706.7	UMC 130 µm
[GSM17]	2	34.6	75	1 648	1 951.2	UMC 130 µm
[GSM17]	2	232.3	4 800	25	29.8	UMC 130 µm
[SM21b]	2	231.5	0	72	108.0	UMC 130 µm
this	2	325.9	9 600	120	40.8	NanGate 45 nm
this	2	52.2	150	3 144	2 954.9	NanGate 45 nm
this	2	54.1	75	4 680	2 901.4	NanGate 45 nm
this	2	117.7	2 400	264	211.2	NanGate 45 nm

F Comparison to Streamlined NTRU Prime

Land et *al.* [LMRG24] recently presented a gadget-based implementation of Streamlined NTRU Prime. Table 19 compares their implementation to ours of the lower two parameter sets of Kyber-CCA decapsulation. Notably, our DSE is capable of generating designs that are more performant regarding each metric. For the area, it is worth noting that their implementation requires significantly more SRAM.

Table 19. Comparison between our Kyber-CCA implementation and Streamlined NTRU Prime [LMRG24]. All implementations use the NanGate 45 nm library. We compare to Kyber-512 (NIST level 1) and -768 (NIST level 3) only, which offer a similar security like sNTRUp-761 (NIST level 2).

d	Scheme	Area	SRAM	Rand.	Latency	Delay
		[kGE]	[bit]	[bit]	[cycles]	[μs]
1	sNTRUp-761	201	189 440	310	1 870 049	9 034
	Kyber-512	447	28 608	326	36 368 922	136 213
	Kyber-512	454	28 608	163	52 707 120	205 887
	Kyber-512	485	28 608	211	931 632	3 741
	Kyber-512	668	28 608	3 880	138 441	541
	Kyber-768	433	35 776	250	66 103 166	250 391
	Kyber-768	439	35 776	125	95 698 594	359 769
	Kyber-768	469	35 776	173	1 557 154	6 254
	Kyber-768	797	35 776	3 956	209 081	791
2	sNTRUp-761	373	246 272	930	1 870 049	11 334
	Kyber-512	575	30 400	978	36 368 922	137 241
	Kyber-512	590	30 400	489	52 707 120	198 895
	Kyber-512	635	30 400	633	931 632	3 516
	Kyber-512	1 013	30 400	11 640	138 441	543
	Kyber-768	553	38 336	750	66 103 166	265 475
	Kyber-768	564	38 336	375	95 698 594	359 769
	Kyber-768	609	38 336	519	1 557 154	6 155
	Kyber-768	1 179	38 336	11 868	209 081	795