HADES: Automated Hardware Design Exploration for Cryptographic Primitives

Fabian Buschkowski[1], Georg Land[2*], Niklas Höher[1], Jan Richter-Brockmann[1], Pascal Sasdrich[1] and Tim Güneysu[1,3]

[1] Ruhr University Bochum, Bochum, Germany, firstname.lastname@rub.de
[2] Intel Labs, Portland, OR, USA, georg.land@intel.com,mail@georg.land
[3] 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* are still mostly manual, tedious, and intuition-driven processes. 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 emphasizes the evident necessity for tool-assisted Design Space Exploration (DSE) for efficient and secure cryptographic hardware. To address this demand, we present the HADES framework. This tool systematically 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-ranging selection of symmetric and PQC schemes, including the ChaCha20 stream cipher and the PQC standard Kyber. Notably, for these schemes, we present the first hardware implementations featuring arbitrary-order masking.

Keywords: Design Automation · Design Space Exploration · Hardware Implementations · High-order Masking · PQC · Kyber · Dilithium · AES · SPN · ARX

1 Introduction

Cryptography is one of the main pillars of the security architecture in modern digital applications that are predominantly relying on 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, generation of *secure* and *efficient* implementations of classical and emerging cryptographic primitives for a wide range of different devices is essential when implementing advanced cryptographic protocols in complex security architectures.

Hardware implementations are especially preferred in most high-assurance and high-performance application scenarios, due to the large amount of design opportunities and

[*] The author contributed to this work solely during his time at Ruhr University Bochum.

available low-level protections. Particularly modern (asymmetric) cryptographic primitives with their modular structure, often consisting of numerous adaptable sub-components, offer a high number of possible combinations and configurations during implementation. The sharing of primitives or their sub-components between various systems and applications further expands the dimensions of the design space.

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

With this challenge in mind, the ATHENA project [Gaj] provides a first unified and comprehensive Application Programming Interface (API), aiming to enable a 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 in a highly manual process by the development teams since no automated exploration process is available. This still poses a challenge for teams with excellent cryptographic expertise but no background to bring this into hardware.

Research Challenge. Evidently, there is a need for automated tools that assist designers and engineers in implementing and exploring primitives efficiently in hardware. Automating the DSE with a primary view on security goals, 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.

Clearly, efficiency is not the only challenge for cryptographic implementations in hardware and embedded systems. *Physical implementation attacks* are common threats targeting 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 during the execution of cryptographic operations.

Although a number of sound and effective protection mechanisms, such as masking [CJRR99], have been thoroughly studied and are frequently implemented, 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 cryptographic instances and corresponding design parameters.

Consequently, a security-driven and automated design space exploration in order to find *efficient and secure* hardware instances for cryptographic primitives remains an open research challenge.

Our Contribution. In this work, we present a novel approach to model, design, and describe generic and security-aware cryptographic hardware based on the established principles of hardware design, enabling the automated generation and design space exploration of efficient and secure hardware accelerators. Moreover, we transform our concept into a versatile tool which, due to its abstraction and exploration methodology, allows us to generate efficient and side-channel protected hardware instances for any type 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 instantiating the optimal candidate in a Hardware Description Language (HDL).

For the proof-of-concept implementation of HADES, which is publicly available on

GitHub, we follow a modular and highly scalable approach. This includes an extensive library of basic building blocks (so-called *templates*) that we instantiate to build and compose commonly deployed cryptographic primitives under a set of given security requirements, like side-channel protection. To demonstrate its capabilities, we perform a wide range of case studies, generating HDL files for efficient and protected symmetric and PQC primitives. As a further contribution, we are the first to provide arbitrary-order masked Application-Specific Integrated Circuit (ASIC) implementations of Add-Rotate-XOR (ARX) ciphers such as ChaCha20 as well as the PQC standard Kyber that were automatically generated using the DSE of HADES.

Related Work. Wolfs et *al.* [WAM12] present an Electronic Design Automation (EDA) tool that takes a Haskell-based functional HDL as input and performs automated DSE in order to find the best instantiation. The main differences to HADES are that their work neither features side-channel countermeasure considerations whatsoever, nor the ability of performance prediction. More recently, Knichel et *al.* have presented AGEMA [KMMS22], a netlist transformer that achieves automated side-channel security, but contrarily to HADES is not capable of DSE. Finally, several previous works focus on machine-learning-based DSE [SW12, DZZ$^+$18, LLS19, SW20]. Among other drawbacks of these works, they do not focus on cryptographic applications, thus lack security awareness in comparison to HADES. All these related works are compared to HADES extensively in Section 8.2.

2 Preliminaries

In this section, we briefly introduce the Hardware Construction Language (HCL) SpinalHDL, which serves as the essential basis of the proof-of-concept implementation of HADES. Afterward, we provide a summary of the formal background of our security design, including relevant 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 be used to describe the functionality of a hardware circuit at a higher level of abstraction than traditional HDLs such as VHDL or Verilog are capable of. For this, HCLs are equipped with powerful libraries and convenience features to ease the process of describing hardware while still having complete 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 a wide range of options for the parametrization of hardware designs. In order to integrate into the classical hardware design flow, SpinalHDL additionally provides the necessary translation layer to both standard VHDL and Verilog. Established synthesis tools can then be used to process the resulting hardware descriptions further. Through its black-boxing feature, SpinalHDL is additionally capable of including existing HDL Intellectual Property (IP) into a design with minimal effort by only defining the corresponding 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 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 with an XOR operation. A correct and secure sharing is achieved by choosing d shares uniformly at random (say the first d shares) and computing the remaining share x_d by $x_d = x \oplus \bigoplus_{i=0}^{d-1} x_i$. 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.

In order to implement a masked design, it is not sufficient to share all inputs, outputs, and intermediate data. All functions that operate on these values also need to be adapted to work with the new shared representation. For linear functions, this is straightforward by applying them individually to each share. However, adapting non-linear functions is much more challenging and error-prone, even with long-standing experience.

To ease this process, research focused on the development of atomic units, so-called *gadgets*, which are used to replace insecure Boolean gates to achieve protection against SCA. While the developed gadgets are secure against SCA, combining them into a larger circuit does not necessarily yield a secure circuit. As a consequence, the security of gadgets is evaluated under certain composability notions that ensure the resistance of the composed circuit against SCA. Here, the most notable and state-of-the-art composability notion is Probe-Isolating Non-Interference (PINI) [CGLS21], which allows trivial composition of gadgets if those gadgets are PINI as well. Together with the PINI notion, Cassiers et *al.* introduced the concept of Hardware Private Circuit (HPC) and the corresponding HPC1 and HPC2 gadgets. Recently, Knichel and Moradi proposed HPC3, reducing the latency of the multiplication gadget to one clock cycle [KM22b].

3 Design Rationale

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 and more difficult to precisely determine the interaction of various parameters and configurations, which gets even more challenging if additional protection against physical implementation attacks is required.

3.1 Problem Definition

To underline the need for automated support in cryptographic hardware design, we recapitulate the common design steps in the development of hardware modules:

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 analysis of the adversary model must be carried out, including potential (physical) attack surfaces, such as side-channel security. This results in additional design goals on the targeted level of hardware security.

2. **Functional Design**: The developer generates HDL code to implement the desired functionality, potentially following different design approaches (e.g., top-down vs. bottom-up). This manual process usually involves many separate and ad-hoc design decisions – many for symmetric cryptography but even more for asymmetric cryptosystems. Each of those decisions impacts the overall performance metrics as well as potential design choices occurring later in the process.

3. **Synthesis**: The hardware description is synthesized into a netlist. Subsequently, the functional correctness and post-synthesis area and delay can be assessed. However,

a meaningful verification of implementation security is not possible until Step 6. Moreover, even though a first shortcut back to Step 2 can be taken in case functional correctness or the post-synthesis assessment do not yield the desired results, this reiteration already requires a *manual* adjustment of the implementation.

4. **Technology Mapping**: The synthesized design is mapped 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 hardware security can be tested by physical measurements as well as verification tools. Feedback from each validation iteration is used to adjust the HDL code until the design goals are met. Thus, a reiteration from the *second* step is required, most importantly resulting in another manual adjustment of the implementation.

7. **Deployment**: The final design is sent to a foundry where the chip is fabricated.

With this current workflow that relies on HDL-based toolchains, two main challenges remain: First, whether or not design goals are achieved is determined at the very end of the design process, 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* Step 3, the synthesis, which is the first of four subsequent computationally expensive and time-consuming operations in the flow. The goal is, thus, to reduce the number of iterations over Steps 3–6 in order to reduce computational cost and development delays. Second, due to the lack of security-by-design integration into the toolchains, 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 instead provide tools to enable informed decisions. While this is true for hardware development in general, the hardware security requirements of cryptographic modules add another dimension to the design space, magnifying the complexity of decision consequences. Hence, the synthesis step – before which these decisions are made – is the crucial point that we want to pre-empt with our tool in order to reduce the necessary number of iterations over Steps 3–6.

We identified the following features to solve the aforementioned challenges in the development process that are currently not supported with conventional HDLs:

- Integration of security-oriented design considerations (e.g., to defeat physical attacks by including side-channel countermeasures) into the workflow (cf. Section 3.2.2).

- Automated pre-evaluation of different design options *before* synthesis while guaranteeing their functional correctness (cf. Section 4.1).

- Prediction of certain performance metrics *before* synthesis to enable a swift design space traversal given many thousands or millions of design configurations (cf. Section 4.3).

3.2 Templates

To address the above-stated features, we introduce our concept of *nested hardware templates* with intrinsic support for security requirements as a fundamental principle for the

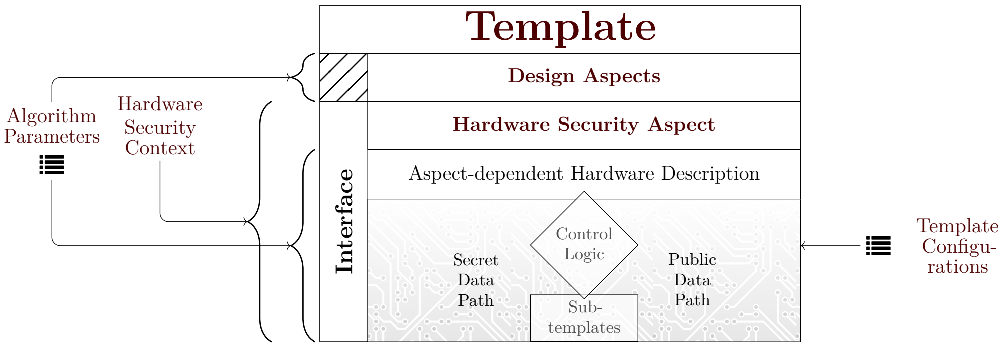


Figure 1. The concept of security-aware hardware templates. Here, a template defines a module implementing specific functionality like an adder. As depicted, the Design Aspects (e.g., the adder type in the case of an adder template) do not influence the interface. In contrast, the Hardware Security Aspect (e.g., gadget type) may change the interface (in this example, the number of shares). The actual hardware description then dynamically depends on the aspects.

abstract modeling and performance prediction of arbitrarily complex hardware circuits for cryptographic primitives. This concept, shown in Figure 1, is explained in this section, while details on template processes and their interactions with template internals are explained in Section 4. In Figure 2, we show an excerpt from the Keccak template as a running example to highlight the different template settings and workflows. To highlight how templates intrinsically support hardware security requirements, we additionally show a small excerpt from the `Template` class, the base for all templates, in Figure 3. It should be noted that template designers do not need to modify this class unless they want to add a new hardware security context or hardware security aspect.

Our hardware templates are characterized by their extensibility and customizability in terms of application-specific algorithm parameters, template configurations, and the hardware security context, as depicted in Figure 1, enabling us to predict a design's 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.

To capture the many different ways how a hardware component can internally achieve the desired functionality and security, our templates can be further customized by design and hardware security aspects (Section 3.2.2). The concrete implementation of a template's functionality is then covered by a hardware description that depends on both of these aspects (Section 3.2.2), potentially utilizing various subtemplates. In contrast to modern HDL component representations, e.g., in terms of entities (VHDL) or modules (Verilog), our concept has the hardware security context closely associated, which particularly enables the generation of secure designs by replacing sensitive functionality with secure counterparts. Through additional template metadata (Section 4.2), our concept intrinsically enables a *performance prediction*, 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 approach promotes 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 any sensitive parts, focuses on creating extensible, customizable, and secure hardware templates with a strong understanding of the various implementation options and their influence on performance

```
case class Keccak(w : Int = 64) extends Template {
  val io = new Bundle { //Interface
    val stateIn = in Vec(Vec(spinal.core.Bits(w bits), 5), 5) //Input
    val stateOut = out Vec(Vec(spinal.core.Bits(w bits), 5), 5) //Output
  }
  var chiParallel : Int = 64 //Parallelism for Chi step
  override def getOptions(): Map[String, Configurable] = Map( //Design aspects
    "parallelism" -> IntegerList(List(1, 2, 4, 8, 16, 32, 64))
  )
  override def configure(options: Map[String, Any]): Unit = { //Configure with Template Configuration
    chiParallel = options("parallelism").asInstanceOf[Int]
  }
  override def instantiate(): Boolean = { //Hardware description
    val chiIn, chiOut = Vec(Vec(Bits(w bits), 5), 5) //Chi input and output

    //Chi
    for (z <- 0 until w) {
      for (x <- 0 until 5) {
        for (y <- 0 until 5) {
          val zOff = (z+chiParallel) % w
          if (z < chiParallel) {
            chiOut(x)(y)(z) := Xor(chiIn(x)(y)(zOff), And(Not(chiIn((x+1)%5)(y)(zOff)),
                  chiIn((x+2)%5)(y)(zOff)))
          } else {
            chiOut(x)(y)(z) := chiIn(x)(y)(zOff)
          }
        }
      }
    }

    val chiLatency = latencyAnd + latencyXor + latencyNot + 1 //Latency of the Chi function
    this.latency = 24 * (1600 / (chiParallel*25) * chiLatency + 3) //Total latency of Keccak
    this.reload = this.latency
    true
  }
}
```

Figure 2. Excerpt from the Keccak template, showing the template settings as well as the hardware description for the χ step. Note that some lines are omitted, for example those containing import clauses.

and security. On the other hand, the template user, liberated from concerns about the functional correctness of the template or its security features, can seamlessly generate a variety of functional and secure designs while using the designed templates.

The modular approach presented above fosters an agile development process while ensuring functional correctness and supporting the integration of security features, essentially moving the hardware development towards security-aware 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 while facilitating a more responsive and efficient development cycle.

3.2.1 Template Settings

During the design process of templates, each template in the hierarchy of the hardware model definition is parametrized and configured with three different kinds of settings, as depicted on the left and right side of Figure 1:

Algorithm Parameters are immutable under DSE since they define the inherent algorithmic functionality and external interface of each template in the cryptographic primitive. For hierarchical hardware models and nested templates, this specifically requires the inheritance of algorithm parameters to extend the parametrization to all child templates consistently. As depicted in the very left part of Figure 1, the algorithm parameters can influence the interface and logic (lower part), as well as the available design aspects (upper part). *Example: The width of the Keccak function,*

```
10  abstract class Template() extends Component with BooleanGadget with GlobalConfigUser {
11    val gadgetConfig = GlobalConfig.config.gadgetConfig
12    gadgetConfig.numShares = gadgetConfig.securityOrder + 1 //Hardware security context
13    gadgetConfig.latencyAnd = 1 //Hardware security aspect: HPC3 gadget
14    gadgetConfig.randomnessAnd = gadgetConfig.securityOrder * (gadgetConfig.securityOrder + 1) //HPC3

130   var latencyAnd = gadgetConfig.latencyAnd
131 }
```

Figure 3. Excerpt from the `Template`-class, highlighting how the hardware security context and aspects influence template internals.

which is defined in Line 9.

Template Configurations are application-independent and mutable under DSE. They solely define how template-internal data processing mechanisms are implemented in order to realize the intended functionality. Consequently, template configurations do not affect the external functionality or interface of the given template to ensure consistency throughout the DSE process. *Example: The number of concurrently executed χ functions for Keccak. The different options for the parallelism, defined in Line 16, modify how the χ step is implemented in Line 47.*

Hardware Security Context: This defines a security-aware design flow by describing the security level of sensitive data processing or control parts within the template. As shown in the left part of Figure 1, it may impact the interface because data might be passed and returned redundantly, as well as the available hardware security aspect. Similar to algorithm parameters, the hardware security context is immutable under DSE. *Example: The masking degree to counter side-channel attacks or other parameters related to the implementation security, such as the amount of redundancy to counter fault-injection attacks. In Figure 3, the hardware security context is the masking degree, which impacts the number of shares in Line 12 as well as the amount of randomness needed for logic gates in Line 14.*

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

Moreover, our approach significantly differs from the traditional development process with 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-independent requirements during DSE. Our approach, on the other hand, not only allows for a dynamic adjustment of configurations during DSE but also ensures that the template user's input is largely limited to 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 Template Internals

Recall that we differentiate between the template developer[1] and its user. The former defines the behavior of a template depending on its algorithm parameters, template configurations, and hardware security context. As this process shares some similarities to

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

writing hardware descriptions with generics in a traditional HDL, experienced hardware developers can seamlessly adapt to our new workflow.

To enable an efficient DSE with functional and security-aware hardware modules as a 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 design aspect then has several *aspect options*, of which one can be chosen. For example, the parallelism degree of a template could be a design aspect, with a set of integers as its associated options. In our running example, one design aspect (parallelism) is defined in Line 16, with the aspect options being the first powers of two. Design aspects do not interfere with the interface nor the high-level functionality of the template but solely change how the intended functionality is internally achieved. On the other hand, however, the design aspects and their options may be impacted by the algorithm parameters, e.g., the available range of unrolling might depend on the Advanced Encryption Standard (AES) security level. In Figure 1, this duality can be observed by the fact that the algorithm parameters are shown as input to the design aspects, but the design aspects are placed disjoint to the interface.

Hardware Security Aspect: Similar to the design aspects, different *aspect options* to instantiate the security-sensitive parts may be defined depending on the given hardware security context. For example, if the security context defines a masking degree, multiple types of gadgets to protect the secret data path may achieve the same security level but can have different performance characteristics. Then, each gadget type is a potential option for the hardware security aspect. Figure 3 shows how the chosen hardware security aspect, in this case the HPC3 gadget, changes the latency of logic gates (Line 13) and the needed randomness (Line 14) (we omit similar lines for the other types of logic gates). The HPC2 and HPC3 gadgets are predefined hardware security aspects in the `Template` class, but further gadget types can be added by template designers. In Figure 2, the chosen hardware security aspect influences the calculation of the χ step in Line 48 as well as the definition of the latency in Line 70 due to the different latencies of masked gadgets.

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 design 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. An excerpt of the hardware description of the Keccak template, namely the χ step, is shown in our example starting from Line 21. The selected parallelism is utilized in Line 47, limiting the total number of χ operations that are performed in parallel. If a combination cannot be instantiated, e.g., due to mutually exclusive options or because an option is not yet implemented, the designer flags it as a conflict by returning `false` in Line 73. A conflict will be handled by the configuration extraction procedure accordingly.

4 Workflow

As depicted in Figure 4, we separate between the design and exploration flow of templates. In this section, we present further details about the individual flows that arise from this separation and consequently enable the efficient DSE and generation of secure designs.

We want to stress that our concept is not designed as a replacement to standard workflows in hardware development but as an essential extension to accommodate the security

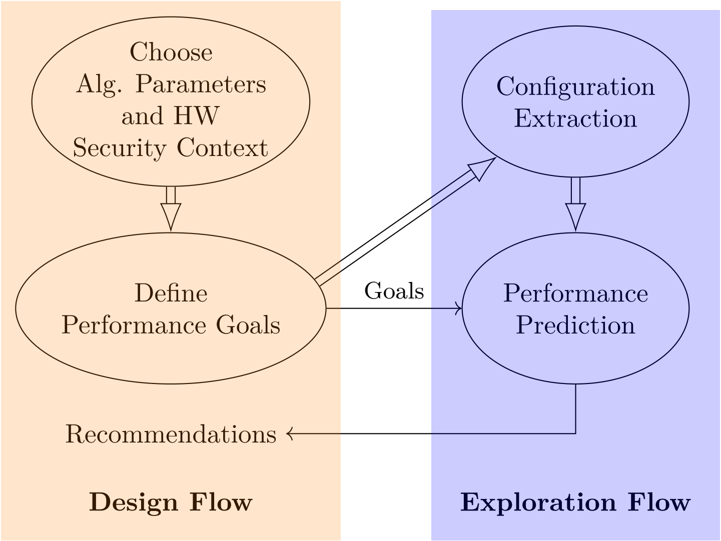


Figure 4. Overview of the design and exploration flows.

context. With the final output being standard VHDL or Verilog, our proposed workflow can be integrated into the established hardware toolchain. In particular, our concept extends Step 2 from the common digital design flow ("Functional Design", cf. Section 3.1) to reduce the necessary iterations that stem from Step 6, the verification and validation.

The design flow (left part of Figure 4) starts with the template user choosing the algorithm parameters that match the current use case. Furthermore, the user specifies the hardware security context according to their threat analysis and attacker model. Finally, the user defines the performance goals and which Performance Metrics (PMs) to optimize and eventually receives the recommendations for the best-performing designs, which can then be further processed by synthesis tools. In the following, we focus on the exploration flow (right part of Figure 4), which is much more complex and covers those areas that we aim to automate with our work.

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 predicted performance among all possible candidates. The DSE for the first approach may terminate as soon as the predicted performance meets all requirements, while the trivial solution for the second approach is exhaustive design space traversal. Yet, more sophisticated methods and, in particular, heuristics may speed up the search for a global optimum or an approximation thereof, which we further discuss in Section 4.4.

Consequently, our concept of nested hardware templates is particularly designed to enable both DSE objectives efficiently. The prerequisite for a traversal of the entire design space is the extraction of all possible design configurations from the template hierarchy. We refer to *design configurations* as the configuration of the entire template hierarchy, as opposed to *template configurations*, which are configurations specific to one template.

4.1 Configuration Extraction

The configuration extraction (cf. Figure 4) is presented in Algorithm 1. Using this algorithm, a comprehensive list of available template configurations and potential dependencies is extracted for each (sub-)template in the template hierarchy, excluding those configurations that were flagged as a conflict by the template designer. This check for conflicts is performed in Line 15. In our running example, the extraction of the available template configurations for the Keccak template is done via a simple function call in Line 15.

Algorithm 1 Configuration extraction procedure. Here, $P_{a,h}$ is a template parametrized with algorithm parameters a and hardware security context h.

```
1:  procedure ExtractConfigs($P_{a,h}$)
2:      $C := \emptyset$ (the set of configuration tuples)
3:      $D :=$ design aspects of $P_{a,h}$
4:      for all $d \in D$ do
5:          $O_d :=$ possible options of $d$
6:      $H :=$ hardware security aspects of $P_{a,h}$
7:      $C_P := H \times O_{d_1} \times \ldots \times O_{d_{|D|}}$          ▷ Each element is one configuration
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:             $a' :=$ DeriveAlgParameters$_c(a, h)$          ▷ Defined by designer
12:             $C_s :=$ ExtractConfigs($s_{a',h}$)
13:         $C_c := C_{s_1} \times \ldots \times C_{s_{|S_c|}}$ ▷ Each entry in $C_c$ is a tuple of subtemplate configurations
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 $C$
```

Essentially, Algorithm 1 traverses a decision tree of configuring templates and their subtemplates. This tree walk is depicted exemplarily in Figure 5. Note that the tree in Figure 5b 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. The algorithm parameters of the subtemplates are, on the other hand, 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.

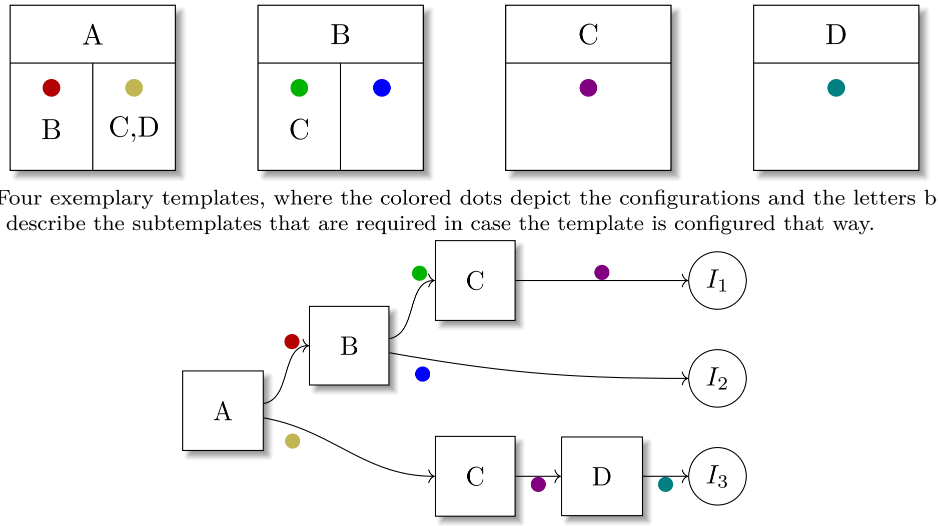
4.2 Performance Prediction

Prediction of PMs is a fundamental process required to guide the DSE and evaluate the suitability of different design configurations of a cryptographic primitive with respect to the specified design requirements. To achieve this, each template predicts various PMs based on its algorithm parameters, template configuration, and the hardware security context, as depicted in the bottom part of Figure 4.

For appropriate choices of PMs and their corresponding prediction, we identified the following requirements and conditions:

Accuracy: The PM prediction needs to be reliable in order to correctly select design configurations 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 the utilization of minimal data on design characteristics



(a) Four exemplary templates, where the colored dots depict the configurations and the letters below them describe 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 5a is used as the top-level template.

Figure 5. An exemplary depiction of templates, configurations, and how configurations are extracted by Algorithm 1.

(while still ensuring accuracy). To emphasize the need for high efficiency, consider our case study of Kyber-CCA, which has over one million possible configurations (cf. Table 10). If estimating the PMs for each configuration took one second, the whole DSE process would take nearly two weeks to complete, which is not feasible in most cases. We further discuss this in Section 8.1.

Coverage: The more hardware module characteristics, e.g., latency, throughput, area utilization, or power consumption, are taken into account, the better a holistic exploration under consideration of various design trade-offs is possible. As the inclusion of more characteristics generally results in a less efficient performance estimation, a careful selection of the most important and meaningful characteristics for the given use case is necessary. For instance, while the power consumption might be critical for a design targeting a constrained device, it can potentially be ignored in a different use case where it is of lesser importance.

While PMs can generally be chosen and defined by the user, a cycle-accurate latency metric is essential in order to generate functional designs and guarantee their correctness. In particular, due to our nested hardware template model, modifications to the template configuration of subtemplates may also impact the latency of the parent template. Therefore, cycle-accurate latency specification is mandatory and must be implemented in any case. In our running example, the latency of the template is specified in Line 70 and Line 71, where it is calculated based on the selected parallelism, the latency for a single execution of the χ function (which depends on the latency of different logic functions, which in turn depends on the hardware security context and the selected option for the hardware security aspect), and some constants.

Similarly, we require a template to report a cycle-accurate reload, i.e., the number of cycles to wait until the next input is accepted. This allows for precise scheduling and ensures that data can be fed into the system at the correct intervals without causing data hazards or stalls. In our running example, the reload is specified in Line 72. It is equal to the latency as the Keccak design is not pipelined and can only accept new inputs once the previous computation is completed. By enforcing a cycle-accurate reload metric, we can avoid issues such as pipeline bubbles or under-utilization of hardware resources. Both the latency and reload metrics are crucial in optimizing the overall design and ensuring the system meets its performance and functional requirements.

When chosen accordingly, the PMs 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 subtemplate prioritization or iterative subtemplate refinements (see Section 4.4). Particular examples for PMs include, apart from the aforementioned cycle count latency, the area, power consumption, or critical path delay.

4.3 Performance Prediction Accumulation

Based on the generated list of 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 their parents' configuration. In our running example, the template is configured with the currently selected parallelism in Line 18, overwriting a local variable that is used in the hardware description of the template.

Upon completion of design configuration, the templates estimate, accumulate, and report the user-specified PMs in a bottom-up process, i.e., starting with the lowest level until the top-level template is reached, yielding the final performance prediction. To accumulate the PMs of a template and its subtemplates, each template either provides a customized aggregation function defined by the template designer or uses simple accumulation of PMs as the default procedure.

The default accumulation procedure for the area and randomness PMs of a template is shown in Algorithm 2. The recursive algorithm is first executed for all subtemplates in Line 5, returning the area and randomness PM for each subtemplate, which are then added to the accumulators of the current template. Afterwards, Algorithm 2 estimates the area and randomness for the hardware description of the current template. For this, the procedure iterates through all logic gates (e.g., AND, XOR, NOT, cf. Line 48 in our running example) and registers that are used in the template and counts their occurrences. Afterwards, the number of occurrences for each gate type is multiplied with its approximate size or randomness demand and added to the accumulators.

If the DSE objective is to find a design that fulfills specific performance requirements (i), either the current configuration meets these requirements, or the DSE procedure continues until a suitable candidate has been found. In case the user wants to find the optimal design across all possible candidates (ii), the design space is explored exhaustively and the best configuration is identified. 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.

4.4 Efficiency

Searching the design space exhaustively may result in impractical run times since the design space grows exponentially in the depth of templates. In this context, it is important to recall that the configuration extraction is expected to be very efficient, as it is a simple graph walk, while performance prediction on the other hand is computationally more significant. Therefore, several measures can be taken to reduce the run time complexity.

Algorithm 2 Performance accumulation procedure for the area and randomness of a template P.

```
procedure AccumulatePerformance($P$)
    $A := 0$
    $R := 0$                                   ▷ Initialize area and randomness count with 0
    $S :=$ subtemplates of $P$
    for all $s \in S$ do
        $(A_s, R_s) :=$ AccumulatePerformance($s$)
        $A := A + A_s$
        $R := R + R_s$                         ▷ Accumulate area and randomness for every subtemplate
    for all $l \in L$ do                       ▷ $L$ is a list of all gate types used in $P$
        $n_l :=$ CountGates($P, l$)            ▷ Count the number of occurrences of gate $l$ in $P$
        $A := A + n_l \cdot$ Size($l$)
        $R := R + n_l \cdot$ Randomness($l$)
    return $(A, R)$
```

4.4.1 Breakpoints

To break the exponential growth, we can apply a divide-and-conquer approach to the DSE. For this, the user defines the maximum number of configurations $c_{\max}$ that shall be explored exhaustively. Once the configuration extraction crosses the threshold of $c_{\max}$ configurations, it is relaunched from a different point in the template hierarchy. Specifically, given a current template hierarchy depth of d, it is relaunched independently for all templates that sit a depth $\lfloor d/2 \rfloor$. This splitting is repeated until the total number of configurations does not exceed $c_{\max}$ anymore.

Subsequently, performance prediction is carried out separately on each resulting split part of the template hierarchy with new depth $\lfloor d/2 \rfloor$ in bottom-up direction. Only the best-performing candidate for each of these parts, according to the target metric, is then passed to the respective top-level template.

It is important to note that for non-combined PMs, this approach is as accurate as an exhaustive search. For example, choosing options with the lowest area footprint bottom-up will always result in the smallest overall design. Conversely, this approach is more inaccurate and rather a trade-off between accuracy and DSE efficiency for combined metrics like the area-latency product, where the previous argument does not hold.

4.4.2 Local Search

Local search algorithms [TC23] are a family of metaheuristic algorithms that are frequently used in optimization problems as they are typically able to quickly find a local optimum while being simple and easy to implement. To allow finding optimal configurations with regard to combined PMs, we make use of the simplest form of local search, starting from a (very likely) non-optimal configuration and altering single design aspects until we reach at least a local optimum.

In order to achieve this, we first need to distinguish between design aspects with well-ordered options (usually parallelism options) and those without (e.g., different ways to implement adders). We assume that the well-ordered aspects have a monotone impact on individual performance metrics. The non-well-ordered aspects, on the other hand, first need to be ordered to prepare for the local search. This is achieved by first predicting the combined PM for a user-configurable number of randomly sampled configurations, and then storing the achieved minimum PM for each aspect option. These minima are then taken as keys to order the respective aspect options.

After ordering the options of all design aspects, we conduct the local search as follows:

1. We start with the configuration from the previously selected sample set that offers the overall best combined PM value as it is the most promising to be close to a local minimum. This PM has already been predicted in the preparation step.

2. Next we iterate over all design aspects from the currently selected design configuration.

3. For each design aspect, we predict the performance for the options that are adjacent to the currently selected option. Adjacency is determined via the given order for the well-ordered aspects or via the estimated order for the non-well-ordered aspects.

4. If one of the performance predictions has improved the targeted combined PM, we use this design as our new current configuration and continue improving the next design aspect.

5. Once all aspects in the hierarchy have been processed, we check whether at least one has changed; if not so, we stop and return the current design configuration as we have found a local optimum; otherwise, we return to Step 2 and start over with the first aspect.

Evidently, this leads to a local optimum and diminishes the number of performance predictions that have to be carried out. Whether the *local* optimum is also the *global* optimum consequently highly depends on the number of initial samples. Increasing this number leads to a more precise ordering of the non-well-ordered aspects as well as a better starting point for the local search itself. Clearly, however, the number of performance predictions grows when opting to increase the number of initial samples. Eventually, our practical evaluation (cf. Section 8.1) shows that this general instance of local search finds the global optimum in many cases, even with small sample sizes, significantly improving the DSE performance.

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 implementation of 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 trivially allows for the implementation of templates (i) as well as the DSE (iii). In particular, when realizing templates as classes, the template hierarchy can be represented via the instantiation of sub-classes, even when multiple subtemplates of the same type are required. Using the inheritance functionality of OOP, algorithm parameters can simply be propagated from the top-level template to its subtemplates. Furthermore, template internals, such as the design aspects with their possible configuration options or the PMs, can be stored as fields inside the template class. This enables a straightforward configuration extraction and template configuration during the DSE through dedicated class methods and further allows us to access and modify the PM during the performance prediction.

Consequently, we decided to use a HCL 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 a convenient 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 a suitable choice as it is among the most feature-rich and well-maintained HCLs that can be augmented with additional properties. Furthermore, the translations to standard VHDL and Verilog ensure that the generated designs can be further processed by the established toolchain for synthesis and placement. Chisel [Chi23], an alternative HCL embedded into Scala, was also investigated regarding its suitability. 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 from a high-level language, such as C, into a hardware description. This approach would allow for a quick implementation of designs through the rich features of high-level languages and require little knowledge about the underlying 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 performance, especially for complex cryptographic designs. Certain design aspects, such as an unrolling or pipelining factor, are furthermore often not chosen by the designer but instead determined by the HLS tools, limiting the number 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 constructor parameters of the template class. This also allows potential sub-classes to inherit algorithm parameters from the parent class. The hardware security context, which is not only immutable but also identical for all templates in our proof of concept, is implemented as an object that is created at the start of the DSE and shared between all templates. The configurable design aspects and the hardware security aspects are realized as static fields in the template class. This allows the design options to depend on algorithm parameters (cf. Section 3.2.2) and enables access to these 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 of our proof-of-concept implementation of HADES currently supports four PMs, namely *Latency*, *Reload*, *Area*, and *Randomness*. While the latency and reload of a template are computed cycle-accurately based on the information specified and stored in the template by the template designer (cf. Line 71), the area of an instantiated template can only be estimated. This estimation is based on the number of registers and logic templates used in a template and a list containing the sizes of registers and logic templates 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 extended for arbitrary cell libraries. The randomness demand of a design can be accurately calculated based on the amount and type of logic gates used in each template as well as 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, our proof-of-concept implementation additionally calculates the *Area-Latency Product (ALP)* and *Area-Latency-Randomness Product (ALRP)* by multiplying the respective PMs, yielding balanced designs. Our implementation 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.

To perform the DSE, our implementation first extracts a list of all possible design configurations according to Algorithm 1. For every found design configuration, the PMs are accumulated using a bottom-up procedure. The PMs of the top-level template, equivalent to the performance of the entire design, are stored in a table together with the corresponding design configuration. Sorting this table by a certain PM allows us to find the optimal design configuration in the respective category. In case of a tie between multiple design configurations (which can typically happen for latency or randomness), the table can be further sorted by a secondary PM to find the best-suited design. Users can additionally specify thresholds for one or multiple PMs that configurations must not exceed. Then, the DSE is performed as before, but only it returns configurations within the specified thresholds, or an error if no suiting configuration was found.

6 Case Studies

In this section, we showcase the performance and versatility of our tool and perform DSE on various common cryptographic components and schemes. Notably, we provide implementations of primitives with known implementations in the literature, but also novel implementations, e.g., for ChaCha20 and Kyber, which demonstrates the adaptability of our tool.

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

For the synthesis, we use Synopsys Design Compiler, version S-2021.06-SP4, with the NanGate 45 nm library. We do not perform an extensive synthesis optimization but rather set the timing goal to 10 µs, which is typically surpassed by the synthesis tool. It should be noted that the reported performance numbers may be further optimized by a more sophisticated synthesis routine. In order to compute the delay values presented in this Section, we multiply the latency of a template in cycles with the critical path delay as given in the synthesis report. Due to the long synthesis times for some of our designs, we abort the synthesis after 72 hours and highlight the affected designs in the following tables.

Throughout this Section, we use various abbreviations in the tables: Opt. (optimization target), Design Config. (design configuration), Rand. (randomness, in bits per cycle), Lat. (latency, given in cycles), est. (estimated), and T.put (throughput). We further use abbreviations that are specific to case studies: QR (quarter round), pipel. (pipelined), para. (parallelism), arch. (architecture), add. (adder), and comp. (comparison).

6.1 Efficient Addition

Efficient addition has been thoroughly scrutinized 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 category of parallel prefix adders achieves a better critical path delay by performing 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.

Table 1. DSE and synthesis results for an adder up to the second masking degree. Exemplarily, we choose 16- and 32-bit widths, but this value 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

In the field of side-channel security, Schneider et *al.* [SMG15], for the first time, introduced 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 foundation of various cryptographic primitives and higher-level templates, we 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, a cycle count latency is induced by the secure gadgets and, thus, a fully-pipelined design arises, 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 trade-offs 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.

Table 2. DSE and synthesis results for modular addition up to the second masking degree. Exemplarily, we choose the moduli of Kyber and Dilithium, but the template allows for the 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	1434	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

6.2 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 an adder template that supports freely configurable moduli 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 utilizes 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 type is used. Table 2 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 is due to the fact that the serial RCA uses control logic and a stage register, which have a relatively high impact on the area for smaller moduli. For larger moduli, the size of the control logic and the state register increases only slightly for the serial RCA, while the additional generate/propagate groups significantly increase the size of the parallel prefix adders.

6.3 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 operations. Notably, the latter two can be performed share-wise for PINI gadgets such as the HPC gadgets. As a representative of this type of cipher, we have exemplarily implemented a template for ChaCha20, configurable for the architecture (round-based or unrolled), adder type (including serial or pipelined), number of quarter rounds to be performed in parallel (1, 2, 4, or 8), and pipelining of the quarter rounds.

Table 3 shows the results of our DSE and the synthesis. The unrolled and fully-pipelined architecture is only selected when optimizing purely for latency or reload since it introduces a significant area overhead that outweighs the savings in terms of latency in

Table 3. DSE and synthesis results for ChaCha20 up to the second masking order.

d	**Opt.**	**Design Config.**					**Area**		**Rand.**	**Lat.**	**Delay**
		architecture	# QR	QR pipel.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
0	L	unrolled	8	true	SKA	—	192.3	176.8	—	10	0.06
	A	round	1	false	SKA	—	3.5	10.9	—	415	0.88
	ALP	round	4	false	SKA	—	5.0	15.4	—	103	0.22
1	L	unrolled	8	true	SKA	HPC3	7335.6	6755.5	102144	486	0.23
	A	round	1	false	sRCA	HPC3	14.7	22.0	4	22576	52.2
	R/ARLP	round	1	false	sRCA	HPC2	14.8	22.0	2	33328	75.0
	ALP	round	1	true	SKA	HPC3	95.1	99.0	1216	570	1.1
2	L	unrolled	8	true	SKA	HPC3	16977.9	n/s*	306432	486	n/s*
	A	round	1	false	sRCA	HPC3	32.5	32.3	12	22576	49.9
	R/ALRP	round	1	false	sRCA	HPC2	32.7	32.4	6	33328	70.0
	ALP	round	1	true	SKA	HPC3	219.4	178.8	3648	570	1.2

*not synthesizeable within 72 hours

all implemented combined metrics. Nonetheless, this architecture is able to offer vastly improved throughput compared to all other configurations, with the first-order masked variant running at over 2.1 GHz and thus exceeding a throughput of 1 Tbit/s.

6.4 Keccak

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

The permutation consists of five steps: the first three (θ, ρ, π) are linear in the Boolean masking domain, the χ step is a quadratic function and requires non-linear gates, and the final ι step is an affine function. The main parameter to explore during DSE, particularly for masked designs, is the number of parallel χ operations to perform. 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 linear steps are carried out simultaneously within a single clock cycle, and the outcome is then stored in the state register. Afterward, 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 trade-off exists between low latency (many χ operations in parallel) and low area and randomness (few χ operations). The trade-off is noticeable in the results of the DSE shown in Table 4.

6.5 Advanced Encryption Standard (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 encryption core is parametrizable for all standardized key sizes, namely 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 (currently Canright or Boyar-Peralta S-Box; different choices are possible), the number of S-Box instances within the key schedule (1, 2, 4, 8), and the number of

Table 4. DSE and synthesis results for Keccak-f[1600] up to the second 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	—	1608	916.76
	ALP	32	—	9.8	19.3	—	120	115.16
1	L	64	HPC3	131.6	149.4	3200	120	40.80
	A	1	HPC3	15.1	30.1	50	3144	1792.47
	R/ARLP	1	HPC2	15.9	34.6	25	4680	3744.00
	ALP	16	HPC3	42.8	61.6	800	264	198.05
2	L	64	HPC3	299.3	325.9	9600	120	40.80
	A	1	HPC3	24.3	52.2	150	3144	2954.89
	R/ARLP	1	HPC2	26.2	54.1	75	4680	2901.43
	ALP	16	HPC3	89.8	117.7	2400	264	211.20

parallel S-Boxes (0, 1, 2, 4, 8, or 16) and MixColumns instances (1, 2, or 4) in a round. If the number of round S-Boxes is set to 0, the S-Box instances dedicated to the key schedule are reused. Table 5 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 halves the latency compared to the round-based ALP optimization while increasing the area by a factor of at least 36.

6.6 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 Number-Theoretic Transform (NTT) or Karatsuba, which are algorithmically faster but infeasible to use with Boolean masking due to the large intermediate coefficient multiplications.

The template we implement follows the schoolbook strategy to enable Boolean masking 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 6 contains the DSE results for the polynomial multiplication for the Kyber-512 use case.

Table 5. DSE and synthesis results for AES up to the second masking degree.

d	**Opt.**	**Design Configuration**							**Area**		**Rand.**	**Latency**	**Delay**
		arch.	SBox	#rSBox	#kSBox	#MC	pipel.	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[ns]
Algorithm Parameter: AES-128													
0	L/ALP	round	Can	16	4	4	true	—	8.6	9.8	—	11	25
	A	round	Can	0	1	1	true	—	2.1	4.2	—	211	597
1	L	unrolled	Can	16	4	4	true	HPC3	1057.4	707.9	14400	41	60
	A	round	BP	0	1	1	false	HPC3	9.5	10.3	72	1011	1446
	R	round	BP	0	1	1	false	HPC2	10.7	11.2	34	1811	2608
	ALRP	round	BP	0	1	1	true	HPC2	13.7	12.7	34	371	575
	ALP	round	Can	4	1	1	true	HPC3	28.9	23.7	360	81	151
2	L	unrolled	Can	16	4	4	true	HPC3	2375.6	1426.1	43200	41	64
	A	round	BP	0	1	1	false	HPC3	20.9	17.2	204	1011	1526
	R	round	BP	0	1	1	false	HPC2	23.5	19.3	102	1811	2898
	ALRP	round	BP	0	1	1	true	HPC2	30.4	21.5	102	371	545
	ALP	round	Can	4	1	1	true	HPC3	64.7	44.6	1080	81	134
Algorithm Parameter: AES-192													
0	L/ALP	round	Can	16	4	4	true	—	9.0	11.5	—	13	32
	A	round	Can	0	1	1	true	—	2.5	6.0	—	253	756
1	L	unrolled	Can	16	4	4	true	HPC3	1307.6	835.3	16128	49	74
	A	round	BP	0	1	1	false	HPC3	11.1	13.8	72	1213	1735
	R	round	BP	0	1	1	false	HPC2	12.1	14.7	34	2173	3064
	ALRP	round	BP	0	1	1	true	HPC2	15.2	16.2	34	445	672
	ALP	round	Can	4	1	1	true	HPC3	30.4	27.3	360	97	147
2	L	unrolled	Can	16	4	4	true	HPC3	2937.0	1660.9	48384	49	78
	A	round	BP	0	1	1	false	HPC3	24.2	22.5	204	1213	1941
	R	round	BP	0	1	1	false	HPC2	26.8	24.5	102	2173	3281
	ALRP	round	BP	0	1	1	true	HPC2	33.8	26.8	102	445	663
	ALP	round	Can	4	1	1	true	HPC3	68.0	49.9	1080	97	147
Algorithm Parameter: AES-256													
0	L/ALP	round	Can	16	4	4	true	—	9.4	6.4	—	15	43
	A	round	Can	0	1	1	true	—	2.9	12.0	—	295	811
1	L	unrolled	Can	16	8	4	true	HPC3	1967.7	1192.2	20160	57	82
	A	round	BP	0	1	1	false	HPC3	12.5	14.6	72	1415	2703
	R	round	BP	0	1	1	false	HPC2	13.6	15.6	34	2535	4664
	ALRP	round	BP	0	1	1	true	HPC2	16.7	17.1	34	519	810
	ALP	round	Can	4	1	1	true	HPC3	31.8	28.1	360	113	163
2	L	unrolled	Can	16	4	8	true	HPC3	4422.5	2298.2	60480	57	86
	A	round	BP	0	1	1	false	HPC3	27.5	23.7	204	1415	2137
	R	round	BP	0	1	1	false	HPC2	30.1	25.8	102	2535	4563
	ALRP	round	BP	0	1	1	true	HPC2	37.0	28.0	102	519	716
	ALP	round	Can	4	1	1	true	HPC3	72.3	51.1	1080	113	147

Sparse Multiplication

We extend the concept explained above to another relevant scenario: 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, **?**], 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.

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

Table 6. DSE and synthesis results for polynomial multiplication with small secrets up to the second masking degree. 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 B (Table 16).

d	Opt.	Design Config.			Area		Rand.	Latency	Delay
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
0	L	256	SKA	—	450	n/s*	—	1281	n/s*
	A	1	sRCA	—	40	94	—	2490625	4408
	ALP	43	SKA	—	108	1192	—	2561	16
1	L	256	SKA	HPC3	7647	n/s*	102400	5121	n/s*
	A	1	sRCA	HPC3	172	142	220	4522241	9725
	R	1	sRCA	HPC2	178	148	110	6553857	12319
	ALRP	1	pRCA	HPC2	192	178	158	81921	144
	ALP	8	SKA	HPC3	391	344	3200	13057	23
2	L	256	SKA	HPC3	17621	n/s*	307200	5121	n/s*
	A	1	sRCA	HPC3	386	193	660	4522241	8004
	R	1	sRCA	HPC2	403	204	330	6553857	11600
	ALRP	1	pRCA	HPC2	435	248	474	81921	144
	ALP	8	SKA	HPC3	893	580	9600	13057	25

*not synthesizeable within 72 hours

Table 7. DSE and synthesis results for 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) up to the second masking degree. Exemplarily, we focus on the Dilithium-2 use case with sparsity $\tau = 39$ and secret range $\eta = 2$. For more results, we refer to Table 17.

d	Opt.	Design Config.			Area		Rand.	Latency	Delay
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
0	L	256	SKA	—	24	n/s*	—	79	n/s*
	A	1	SKA	—	1	16	—	10024	6.8
	ALP	16	SKA	—	5	197	—	664	0.5
1	L	256	SKA	HPC3	1096	4601	13312	274	1.0
	A	1	RCA	HPC3	3	24	36	10375	16.7
	R/ALRP	1	RCA	HPC2	5	28	18	10726	17.3
	ALP	2	RCA	HPC3	4	48	72	5383	10.9
2	L	256	SKA	HPC3	2522	n/s*	39936	274	n/s*
	A	1	RCA	HPC3	6	35	108	10375	19.4
	R/ALRP	1	RCA	HPC2	10	39	54	10726	20.0
	ALP	2	RCA	HPC3	8	67	216	5383	10.8

*not synthesizeable within 72 hours

with it because 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 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 reduction can be performed 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.

The template for our sparse multiplication is configurable for the reduction polynomial (but currently only supports polynomials of the form $x^n \pm 1$), the sparsity, and the range of the secret. The template configuration consists of the adder type and the number of adders that are instantiated. Synthesis results are featured in Table 7.

6.7 Full Kyber Implementation

Kyber is a Key Encapsulation Mechanism (KEM) that serves as the basis of the ML-KEM scheme, which has recently been standardized by the NIST as a PQC alternative to existing key establishment schemes. 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 A.

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 decided for each specific use case whether a CPA decryption is sufficient or if the Chosen Ciphertext Attack (CCA)-secure decapsulation is required. Indeed, Düzlü et *al.* presented a lightweight identification protocol [DKPS23], which requires side-channel security only for the CPA decryption.

Consequently, we provide implementations for both cases. Table 8 shows the DSE and synthesis results for CPA decryption. Our template for decryption features a fully configurable polynomial multiplication avoiding the usage of the NTT in order for gadget-based masking to remain feasible. Additionally, the template has a configurable number of parallel compression steps. Algorithmically, the template can be instantiated for each Kyber parameter set separately but also such that all three parameter sets are supported simultaneously.

For CCA decapsulation, Table 9 shows our DSE and synthesis results. Remarkably, our template structure for Kyber-CCA yields more than *one million* valid design choices, each of which has been explored. Moreover, it is important to note that the smallest designs always work on *a single full adder* that performs all arithmetic.

Table 8. DSE and synthesis results for Kyber-512-CPA decryption up to the second masking degree. Results for the other parameter sets and a design with runtime-configurable security level can be found in Appendix B (Table 18).

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*	—	3333	n/s*
	A	1	1	sRCA	—	42	120	—	4982148	9.03
	ALP	20	32	SKA	—	93	247	—	6929	0.02
1	L	256	256	SKA	HPC3	7737	n/s*	104448	11016	n/s*
	A	1	1	sRCA	HPC3	179	175	228	9045511	18.73
	R	1	1	sRCA	HPC2	186	180	114	13108746	25.45
	ALRP	1	1	pRCA	HPC2	200	211	162	164874	0.29
	ALP	4	8	SKA	HPC3	399	380	3232	26951	0.05
2	L	256	256	SKA	HPC3	17834	n/s*	313344	11016	n/s*
	A	1	1	sRCA	HPC3	403	233	684	9045511	20.79
	R	1	1	sRCA	HPC2	419	244	342	13108746	28.68
	ALRP	1	1	pRCA	HPC2	451	289	486	164874	0.29
	ALP	3	8	SKA	HPC3	910	642	9672	26973	0.06

*not synthesizeable within 72 hours

Table 9. DSE and synthesis results for Kyber-512-CCA decapsulation up to the second masking degree. Results for all parameter sets that also include the design configurations can be found in Appendix B (Table 19).

d	Opt.	Area		Rand.	Latency	Delay	SRAM
		[est. kGE]	[kGE]	[bit]	[cycles]	[μs]	[bit]
0	L	490	n/s*	—	19612	n/s*	26816
	A	75	316	—	19960068	75321	
	ALP	91	459	—	72036	541	
1	L	7869	n/s*	105922	69220	n/s*	28608
	A	259	447	326	36368922	136213	
	R	268	454	163	52707120	205887	
	ALRP	282	485	211	931632	3741	
	ALP	505	668	3880	138441	541	
2	L	18095	n/s*	317766	69220	n/s*	30400
	A	544	575	978	36368922	137241	
	R	564	590	489	52707120	198895	
	ALRP	596	635	633	931632	3516	
	ALP	1113	1013	11640	138441	543	

*not synthesizeable within 72 hours

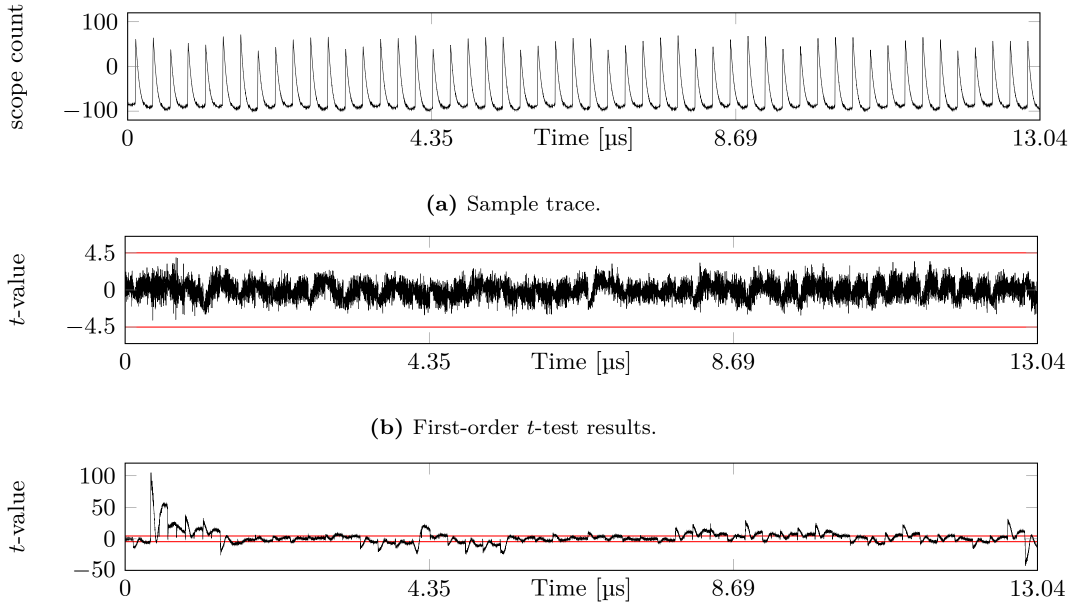
7 Security Evaluation

In order to demonstrate that the masked designs generated by our proof-of-concept implementation of HADES are secure against side-channel attacks, we exemplarily perform a Test Vector Leakage Assessment (TVLA) on a first-order secure round-based AES-128 encryption core using HPC3 gadgets. The selected design uses 16 Boyar-Peralta Sbox instances in parallel in order to compute one encryption in 52 clock cycles. We synthesize the resulting Verilog code for the Sakura-X side-channel measurement board, which is equipped with a Kintex 7 Field-Programmable Gate Array (FPGA). To avoid any recombination of shares in the FPGA fabric, we reimplemented the first-order HPC3 gadgets directly using Xilinx primitives, i.e., Look-Up Tables (LUTs) and FDRE registers. Additional care was taken during the placement phase to prevent possible coupling effects between both shares. The measurement board is connected to a ZFL-2000GH+ Low Noise Amplifier (LNA) configured with a gain of 25 dB and a Spectrum M4 oscilloscope (8-bit resolution). The oscilloscope is configured to sample the power consumption with 1.25 GS/s. To provide the required randomness for the underlying HPC3 gadgets of the AES instance, we instantiate a Keccak core with a state width of 1600 bits, serving as a 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 6 presents the corresponding measurement results. For reference, Figure 6a shows a sample trace of a single encryption performed by the protected AES design. Figure 6b 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 significant signs of leakage (see Figure 6c).

8 Discussion and Comparison

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



(a) Sample trace.

(b) First-order t-test results.

(c) Second-order t-test results.

Figure 6. Side-channel measurement results for an AES-128 encryption core protected by HPC3 gadgets for $d = 1$ (100 million traces).

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 10 shows the number of explored 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 10. Number of configurations and corresponding exhaustive 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	384	3.1s
Polynomial Multiplication	1302	7.9s
AES	4032	10.5s
Kyber-CPA	40362	196.5s
Kyber-CCA	1148364	36h

As demonstrated by the runtime of 36 hours for the Kyber-CCA case study, exhaustively exploring the design space quickly becomes infeasible as a growing number of design aspects results in an exponential number of possible design configurations. This could be mitigated by conducting the performance prediction in parallel (e.g., via launching subprocesses). However, our proof-of-concept implementation does not implement this, in favor of exploring more sophisticated DSE methods rather than purely technical solutions. Section 4.4 names two optimizations for the DSE whose effects on the results and runtime of the Kyber-CCA case study we briefly present and discuss in the following.

Breakpoints. The maximum template depth for the Kyber-CCA template is 4, resulting in possible splits at depth 2 with groups of two templates or after each template, thus exploring and optimizing each template individually. For both splitting options, the DSE was able to find the optimal design for latency, area, and randomness but not for the ALP or ALRP, where the predicted best configurations were far off the ideal configuration. Splitting at depth 2 requires the evaluation of 2510 design configurations, which takes 210 seconds, an improvement of more than two orders of magnitude over the exhaustive search. By splitting after every template, the number of evaluated configurations is reduced to 506 with a runtime of just 12 seconds, more than three orders of magnitude faster than the exhaustive search. These results highlight that for simple PMs, optimizing every template individually allows for a fast and yet accurate DSE, even for large design spaces.

Local Search. The size of the initial sample set has a major influence on the runtime and the results of the local search, most notably whether the optimal configuration is found, and if not, how far away from the optimum the found configuration is. To analyze the influence of the initial sample size, we have conducted experiments with initial sample sizes of 1, 10, 50, 100, 500, and 1000 samples for the ALP of Kyber-CCA (we omit the results for the ALRP, which were very similar). As the local search involves randomly selecting the initial sample set, we have carried out 1000 runs for every sample size and averaged the results, which we present in Table 11.

Table 11. Results of the local search optimization for different initial sample sizes. The table includes the average number of evaluated configurations (including the initial evaluation) per run, how often the optimal configuration was found (out of 1000 runs), the average overhead over the optimal design, and the average runtime per run.

Samples	#Conf.	#Opt. Conf.	Overhead [%]	Runtime
1	27	731	0.20	2 s
10	22	961	0.06	3 s
50	71	1000	0	8 s
100	121	1000	0	13 s
500	520	1000	0	81 s
1000	1024	1000	0	200 s

Even with only one initial sample, the optimal configuration was found in 731 out of 1000 runs with an average overhead of only 0.2%. On average, 27 configurations (including the initial sample) had to be evaluated, which was achieved in only two seconds. For a sample set size of 10, the local search is already successful in 961 runs, and the average overhead decreases to just 0.06%. Strikingly, this sample size requires fewer total configuration evaluations compared to just using a single initial sample. This is attributed to the fact that the initial prediction is significantly better with a larger initial sample set, thus requiring fewer optimization iterations afterwards. For sample sizes of 50 and higher, the local search procedure always finds the optimal configuration, highlighting the effectiveness of this optimization for combined PMs. Even for a sample size of 1000, the runtime of 200 seconds is still multiple orders of magnitude below the runtime of the exhaustive search, which emphasizes the gain in runtime efficiency achieved through the local search approach.

We also tested the effectiveness of the local search optimization for our other case studies. Across the board, the optimization was able to find optimal configurations with small initial sample sets within a few seconds of runtime. For all of our case studies, choosing an initial sample set with a size between 0.01% and 1% of the total number of configurations guaranteed to find the optimal configuration. In addition, the necessary relative size of the

Table 12. Comparison of pre-layout FPGA synthesis results to [WAM12] for a 192-bit adder constructed from an adder of smaller bit width. All HADES designs are synthesized using Xilinx XST 14.7 targeting a Virtex-5 XC5VLX20T FPGA.

Optimization Goal	**Tool**	**Design Config.**		**Area**	**Delay**
		adder	bit width	[slices]	[ns]
Area	[WAM12]	Behavioral	8	594	3.12
	HADES	SKA	8	148	3.11
Delay	[WAM12]	Behavioral	16	600	2.82
	HADES	sRCA	8	158	1.94

initial sample set is reduced with a growing number of total configurations for a design. These findings are especially important for even larger designs than the Kyber-CCA case study, where it might be impossible to exhaustively find the optimal solution, thus requiring the confidence that the local search finds optimal configurations.

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 direct comparison. Nonetheless, both automated DSE for use in cryptographic hardware design and the automated generation of secure hardware implementations have previously been explored individually. In this section, we provide comparisons to existing state-of-the-art tools in either area.

Comparison to [WAM12]. Wolfs et *al.* present an EDA tool in [WAM12] that takes circuit descriptions in a Haskell-based functional HDL as input and is capable of performing automated DSE in order to find the best-suited hardware implementation according to specified criteria for a select set of basic cryptographic building blocks, like adders. Similar to HADES, the resulting implementation is then returned in a traditional HDL. While the main difference to our work is the lack of automated security protection capabilities, HADES additionally improves on the given tool in several significant ways.

In order to ensure manageable runtimes of the DSE process, even for large parameter spaces, we use efficiently computable estimated performance metrics at a pre-synthesis stage to select the optimal configuration. The tool by Wolfs et *al.* instead synthesizes each possible implementation using a third-party synthesis tool and relies on the resulting synthesis reports to evaluate the qualities of every design. Even though this is expected to yield more accurate results that are closer to the final characteristics of the resulting hardware designs, it is not feasible to apply this approach to more complex cryptographic primitives, like Kyber, with potentially millions of configurations since the involved synthesis would dramatically increase the runtime of the DSE. Scalability is further weakened by the fact that only a basic exhaustive search over the complete parameter space is supported compared to the more advanced optimization strategies included within HADES (see Section 8.1).

When comparing the performance of the generated designs to functionally equivalent HADES outputs, we are limited to a small set of unprotected designs that are supported by the existing tool. The authors provide pre-layout synthesis results for various configurations of unmasked 192-bit adders targeting a Xilinx Virtex-5 FPGA which we use as a reference for the comparison in Table 12. For the sake of brevity, we only include the two designs with the smallest area footprint and the highest maximum clock frequency for each of the two tools, respectively. The results indicate that HADES is capable of producing designs that require less area while simultaneously offering a higher maximum clock frequency, thus beating the tool by Wolfs et *al.* in both implemented metrics.

Table 13. Comparison to AGEMA. All designs are 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

Comparison to AGEMA. In the realm of design automation for masked hardware, there is only AGEMA [KMMS22], which is a pure netlist transformer, enabling the automatic masking of netlists of unprotected circuits. As AGEMA relies on already synthesized designs, it does not enable DSE as HADES does. Essentially, AGEMA is restricted to a lower design layer on which it only provides a limited subset of functionality. Nevertheless, a comparison is still possible as both tools produce masked designs. For the sake of simplicity, Table 13 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, a pipelined SKA, and a serial RCA, generated from our three addition templates. Each adder has a width of 32 bits, 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 returned by HADES were synthesized again, providing accurate figures for area occupation and critical path delay (cf. Table 13). 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 C (Keccak), and Appendix D (Kyber). Remarkably, Figure 7 exhibits that HADES outperforms AGEMA in relevant use cases while at the same time offering much more variety with respect to the design performance.

Comparison to Machine Learning Approaches. Several previous works explore machine learning for DSE [SW12, DZZ+18, LLS19, SW20]. Notably, these works do not focus on cryptographic hardware, but hardware development in general. Setting this aside, there are still four aspects that set apart HADES from these works:

1. The above works rely on HLS, while HADES does not.
2. The machine learning approach needs to be fed with synthesis output, which renders the whole procedure much less performant than our approach that relies on pre-synthesis prediction.
3. HADES offers security awareness (i.e., masking and respective DSE), which is – understandably – not featured in the above mentioned generic works.
4. The machine learning approach lacks explainability, which HADES offers fully. Specifically, HADES is able to give a full report on each of its steps and design decisions.

8.3 Limitations and Future Work

Performance Metric Selection. As visible in Section 6, the SKA is consistently picked over the KSA as the configuration for adders during DSE. While both adders yield the same cycle count latency, the SKA has a lower area estimate as a tiebreaker, making this choice reasonable for latency optimization. However, synthesizing both adder types – KSA results are shown in Table 14 – 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.

Most notably, we currently do not predict routing cost, which could mitigate the issue in Table 14. One idea to include this as a metric could be to estimate the maximum or average fan-out of each gate in the templates, which correlates to the overall routing cost. Predicting the fan-out of the secret data path would be rather easy, as we only allow a structural hardware description there, and the fan-out of the single gadgets can be trivially determined. For the public data path, on the other hand, this is much more complex, as a behavioral description is allowed. This could be solved via worst-case estimations for single behavioral operations, although this would require the tool to know all potential operations and their worst-case fan-outs. Even then, multiple such operations might be combined, and a rule to combine multiple estimates would need to be found. Overall, we leave this as an interesting open topic for future research.

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 9, 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 8). 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 cost of less efficient performance prediction.

Competitiveness with Handcrafted Designs. Because HADES technically only traverses the design space, its results heavily depend on the provided templates. When focussing on unmasked designs, HADES may yield results that are competitive to handcrafted designs, since the provided template is nothing different from a generically written handcrafted design. This becomes evident when comparing our AES results to the state of the art

Table 14. 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

in Table 15. Strikingly, we have not found any design in academic literature offering a throughput nearly as high as our latency/reload-optimized design. Moreover, our area- and ALP-optimized designs come very close or outperform previous work.

For masked designs (cf. Table 15, Figure 7), on the other hand, we are bound to the usage of gadget-based masking because of its convenient potential for automatization. With the current state of gadgets, this will likely result in less efficient designs compared to handcrafted works, for example those utilizing the *threshold implementation* technique [NRR06]. At the same time, we observe again that our latency/reload-optimized design achieves an unprecedented throughput in academic literature.

Finally, this argument becomes even more clear when looking at the Kyber results from Section 6.7. As our current template is written with the purpose of being efficiently maskable with gadgets (i.e., deliberately avoiding usage of the NTT), a non-masked instantiation of this architecture necessarily results in a comparatively low-performing – in particular: slow – design[1]. However, due to this choice of gadget-maskability, we can have security guarantees, and enable the automated DSE. And, remarkably, it is still the only published fully protected Kyber hardware implementation, and it consists of more than one million different ways to instantiate.

Model Optimization. Table 15 and Figure 7 exemplarily show a comparison with a wide selection [MPL+11, GMK16, CRB+16, SM21, CBR+15] of state-of-the-art AES implementations, including designs that were automatically masked by AGEMA. Strikingly, our gadget-based designs come close within reach even of handcrafted designs, but eventually fall short. Those handcrafted designs manually incorporate various low-level and algorithmic optimization techniques, e.g., the *changing of the guards* methodology [Dae17] that reduces the randomness requirements vastly, which is also reflected in the results. This, however, is not yet reflected by our tool and the DSE, which solely focuses on the *generation* and *exploration* of hardware designs without consideration of design optimization.

Remarkably, our automatically generated designs achieve similar or better performances compared to the ones from AGEMA, except for the area-randomness plane. Notably, area and randomness are somewhat coupled dimensions for gadget-based designs, as a lower area indicates a lower number of gadgets, and a lower number of gadgets usually means less randomness requirements. It is important to note that generating the wide variety of our AES designs took about *five seconds* (cf. Table 10), whereas all other implementations featured in the table do not offer a comparable level of flexibility.

Low-level Optimization. We view HADES as a high-level mechanism that may incorporate several previously published low-level optimization procedures in the future. For example, Compress [CGM+24] offers an automated optimization of gadget usage and ordering. Clearly, this cannot be applied to entire cryptographic designs, as the authors

[1] A potential remedy to this would be to introduce a pre-synthesis switch in the template that checks whether the hardware security context requires masking, and if not so, branching into a design that utilizes NTTs.

Table 15. Comparison with previous work for unmasked and masked AES implementations. Note that our designs and the ones from [KMMS22] are the only ones to use gadget-based masking, while all other designs are handcrafted and manually optimized.

Ref.	Opt.	d	Area	Rand.	Lat.	Delay	T.put	Technology
			[kGE]	[bit]	[cycles]	[ns]	[Gb/s]	
[MPL$^+$11]		0	2.6	–	226			UMCL18G212T3
[SKS12]*		0	58.4	–			1.6	180 nm
[AR18]*		0	29.4	–	46	39	13.130	65 nm
[AR18]*		0	29.4	–	46	58	8.904	65 nm
[KMMS22]		0	3.3	–	227	188	0.679	NanGate 45 nm
[KMMS22]		0	9.9	–	11	20	6.290	NanGate 45 nm
HADES	L	0	85.5	–	11	23	61.000	NanGate 45 nm
HADES	A	0	4.2	–	211	597	0.214	NanGate 45 nm
HADES	ALP	0	9.8	–	11	25	5.195	NanGate 45 nm
[MPL$^+$11]		1	11.1	48	266			UMCL18G212T3
[GMK16]		1	7.6	28	216			UMC 180 nm
[CRB$^+$16]		1	6.7	54	276			NanGate 45 nm
[SM21]		1	7.1	1	246	1537	0.083	UMC 180 nm
[SM21]		1	7.7	0	246	1537	0.083	UMC 180 nm
[KMMS22]		1	33.1	414	3859	9879	0.013	NanGate 45 nm
[KMMS22]		1	10.0	34	2043	4310	0.030	NanGate 45 nm
[KMMS22]		1	52.6	680	99	202	0.634	NanGate 45 nm
HADES	L	1	707.9	14400	41	60	87.070	NanGate 45 nm
HADES	A	1	10.3	72	1011	1446	0.088	NanGate 45 nm
HADES	R	1	11.2	34	1811	2608	0.049	NanGate 45 nm
HADES	ALRP	1	12.7	34	371	575	0.226	NanGate 45 nm
HADES	ALP	1	23.7	360	81	151	0.847	NanGate 45 nm
[CBR$^+$15]		2	18.6	126	276			NanGate 45 nm
[GMK16]		2	12.8	84	216			UMC 180 nm
[KMMS22]		2	17.6	102	2043	5434	0.023	NanGate 45 nm
[KMMS22]		2	131.6	2040	99	237	0.540	NanGate 45 nm
HADES	L	2	1426.1	43200	41	64	82.580	NanGate 45 nm
HADES	A	2	17.2	204	1011	1526	0.084	NanGate 45 nm
HADES	R	2	19.3	102	1811	2898	0.044	NanGate 45 nm
HADES	ALRP	2	21.5	102	371	545	0.235	NanGate 45 nm
HADES	ALP	2	44.6	1080	81	134	0.955	NanGate 45 nm

∗ These papers report gate counts instead of GE.

report that their tool times out after one hour for a design as small as a 32-bit adder. Nevertheless, this procedure could be applied to single templates as a post-DSE step.

As Table 15 and Figure 7 show exemplarily for AES, randomness is one metric that can be improved, as gadget-based implementations are often very randomness-demanding. For this, Feldtkeller et *al.* [FKS$^+$22] presented a technique to re-use certain fractions of randomness across designs that could be automated and integrated into HADES at a single-module layer. Similarly, Knichel and Moradi present a low-randomness gadget that could be integrated as well [KM22a].

Security against Fault and Combined Attackers. Several issues need to be resolved in order to enable fault countermeasures for HADES, such as treating the public data path correctly. However, several existing approaches could be picked up or integrated, for example DOMREP [GPK$^+$21] which offers security against *either* side-channel *or* fault attackers, but not combined security [SBJ$^+$21, SRJB23]. A more suitable solution here could be CINI MINIS [FRBSG22], which was later improved to CPC [**?**].

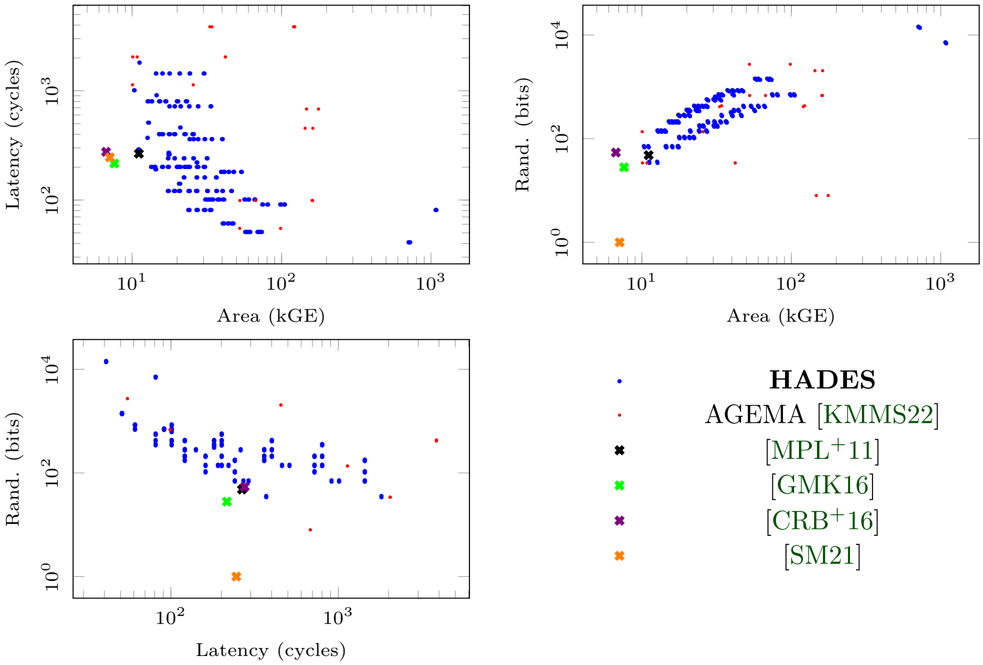


Figure 7. Area-latency, area-randomness, and latency-randomness comparison to related first-order masked AES results. Results closer to the bottom left corner are better. Circle marks indicate that gadget-based masking has been deployed, while cross marks are handcrafted designs.

9 Conclusion

In this work, we present a new framework to systematically explore and instantiate cryptographic hardware, provided by HADES. The accompanying template library covers a wide range of use cases, is easily extendable, reusable, 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 case studies yielded 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 Research, Technology and Space (BMFTR) through the projects VE-HEP (16KIS1345), DI-SIGN-HEP (16KIS2073), 6GEM (16KISK038), EASEPROFIT (16KIS2126), POST (16ME2157), SASPIT (16KIS1859), FlexKI (01IS22086I), and DI-OCDCpro (16ME0942).

References

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

[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.

[AR18] Alexander O. A. Antwi and Kwangki Ryoo. Efficient integrated circuit design for high throughput AES. In James J. Park, Vincenzo Loia, Kim-Kwang Raymond Choo, and Gangman Yi, editors, *Advanced Multimedia and Ubiquitous Engineering - MUE/FutureTech 2018, Salerno, Italy, 23-25 April 2018*, volume 518 of *Lecture Notes in Electrical Engineering*, pages 417–422. Springer, 2018.

[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.

[CGM$^+$24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Generate small and fast masked pipelined circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):500–529, 2024.

[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.

[DZZ$^+$18] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F. Y. Young, and Zhiru Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*, pages 129–132. IEEE Computer Society, 2018.

[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.

[FRBSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: Domain isolation for fault and combined security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1023–1036. ACM Press, November 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.

[GPK$^+$21] Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. Domrep-an orthogonal countermeasure for arbitrary order side-channel and fault attack protection. *IEEE Trans. Inf. Forensics Secur.*, 16:4321–4335, 2021.

[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.

[LLS19] Shuangnan Liu, Francis C. M. Lau, and Benjamin Carrión Schäfer. Accelerating FPGA prototyping through predictive model-based HLS design space exploration. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 97. ACM, 2019.

[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, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, Heidelberg, May 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.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, December 2006.

[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.

[SBJ+21] Sayandeep Saha, Arnab Bag, Dirmanto Jap, Debdeep Mukhopadhyay, and Shivam Bhasin. Divided we stand, united we fall: Security analysis of some SCA+SIFA countermeasures against SCA-enhanced fault template attacks. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 62–94. Springer, Heidelberg, December 2021.

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

[SKS12] PV Sriniwas Shastry, Amruta Kulkarni, and Mukul S Sutaone. Asic implementation of aes. In *2012 Annual IEEE India Conference (INDICON)*, pages 1255–1259. IEEE, 2012.

[SM21] 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.

[SRJB23] Sayandeep Saha, Prasanna Ravi, Dirmanto Jap, and Shivam Bhasin. Non-profiled side-channel assisted fault attack: A case study on DOMREP. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023.

[SW12] Benjamin Carrión Schäfer and Kazutoshi Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *IET Comput. Digit. Tech.*, 6(3):153–159, 2012.

[SW20] Benjamin Carrión Schäfer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(10):2628–2639, 2020.

[TC23] Chun-Wei Tsai and Ming-Chao Chiang. Chapter sixteen - local search algorithm. In Chun-Wei Tsai and Ming-Chao Chiang, editors, *Handbook of Metaheuristic Algorithms*, Uncertainty, Computational Techniques, and Decision Intelligence, pages 351–374. Academic Press, 2023.

[WAM12] Davy Wolfs, Kris Aerts, and Nele Mentens. Design space exploration for automatically generated cryptographic hardware using functional languages. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 671–674. IEEE, 2012.

Appendix

A 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 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. This furthermore reduces the memory footprint of the secret key by a factor of 4.

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

CPA decryption. 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.

We subsequently perform the vector-vector multiplication of $\mathbf{s}$ and $\mathbf{u}$, accumulating this result to the preprocessed v. As a result, 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 3. In addition to the modules required for 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 in parallel to the generation of secret polynomials by the side-channel secure Keccak. Besides, the inverse NTT is required for compliance with the Kyber specification, where, for performance reasons, $\hat{\mathbf{A}}$ is generated in the NTT domain. We contrarily assume $\mathbf{t}$ to not be given in the NTT domain because a key generation module that employs gadget-based masking with Schoolbook multiplication would produce $\mathbf{t}$ that way.

Algorithm 3 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 $K$
```

As indicated in Line 10 of Algorithm 3, 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-masked setting, we instantiate an additional modular addition template 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 be in non-negative modular representation. Consequently, we conditionally add q based on the sign bit of the CBD sample.

B Additional Results

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

- additional parametrizations and third-order masking syntheses for polynomial multiplication (Table 16)
- additional parametrizations and third-order masking syntheses for sparse polynomial multiplication (Table 17)
- synthesis results for Kyber-768 and -1024 CPA decryption and a design that enables runtime configurability regarding the parameter set (Table 18)
- full synthesis results for Kyber CCA decapsulation (Table 19)

In Table 19, we omit the SRAM usage for space reasons. Kyber-512 requires 26816 bit, 28608 bit, and 30400 bit for $d \in \{0, 1, 2\}$. Kyber-768 requires 33216 bit, 35776 bit, and 38336 bit for $d \in \{0, 1, 2\}$. Kyber-1024 requires 53376 bit, 56704 bit, and 60032 bit for $d \in \{0, 1, 2\}$.

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

Table 16. Additional DSE and synthesis results for polynomial multiplication, cf. Table 6.

d	Opt.	Design Config.			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	31697	n/s*	614400	5121	n/s*
	A	1	sRCA	HPC3	688	248	1320	4522241	8004
	R	1	sRCA	HPC2	717	266	660	6553857	11600
	ALRP	1	pRCA	HPC2	774	329	948	81921	144.2
	ALP	8	SKA	HPC3	1599	884	19200	13057	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*	—	1025	n/s*
	A	1	sRCA	—	40	94	—	2425089	3251
	ALP	64	SKA	—	105	186	—	1793	5
1	L	256	SKA	HPC3	6111	n/s*	83968	4609	n/s*
	A	1	sRCA	HPC3	167	138	148	4391169	7772
	R	1	sRCA	HPC2	171	141	74	6357249	8904
	ALRP	1	pRCA	HPC2	184	172	122	81153	135
	ALP	10	SKA	HPC3	390	482	3280	11009	22
2	L	256	SKA	HPC3	14100	n/s*	251904	4609	n/s*
	A	1	sRCA	HPC3	376	184	444	4391169	6583
	R	1	sRCA	HPC2	385	191	222	6357249	9603
	ALRP	1	pRCA	HPC2	414	236	366	81153	106
	ALP	10	SKA	HPC3	890	759	9840	11009	23
3	L	256	SKA	HPC3	25381	n/s*	503808	4609	n/s*
	A	1	sRCA	HPC3	668	234	888	4391169	6633
	R	1	sRCA	HPC2	685	246	444	6357249	9918
	ALRP	1	pRCA	HPC2	738	308	732	81153	107.9
	ALP	10	SKA	HPC3	1595	1108	19680	11009	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*	—	2284	n/s*
	A	1	RCA	—	122	144	—	480644	721
	ALP	254	SKA	—	257	n/s*	—	3806	n/s*
1	L	761	SKA	HPC3	18748	n/s*	307444	12177	n/s*
	A	1	sRCA	HPC3	490	208	82	39960111	60363
	R	1	sRCA	HPC2	492	211	41	57912862	84916
	ALRP	1	pRCA	HPC2	506	454	93	626305	1040
	ALP	35	SKA	HPC3	1194	1306	9870	28158	55
2	L	256	SKA	HPC3	43601	n/s*	922332	12177	n/s*
	A	1	sRCA	HPC3	1101	270	246	39960111	62340
	R	1	sRCA	HPC2	1106	275	123	57912862	87482
	ALRP	1	pRCA	HPC2	1139	543	279	626305	821
	ALP	32	SKA	HPC3	2588	1890	27072	29680	61
3	L	256	SKA	HPC3	78774	n/s*	1844644	12177	n/s*
	A	1	sRCA	HPC3	1958	332	492	39960111	60362
	R	1	sRCA	HPC2	1968	339	246	57912862	90348
	ALRP	1	pRCA	HPC2	2025	631	558	626305	833
	ALP	32	SKA	HPC3	4636	2468	54144	29680	65

*not synthesizeable within 72 hours

Table 17. Additional DSE and synthesis results for sparse polynomial multiplication, cf. Table 7.

d	**Opt.**	**Design Config.**			**Area**		**Rand.**	**Latency**	**Delay**
		# add.	adder	gadget	[est. kGE]	[kGE]	[bit]	[cycles]	[μs]
Algorithm Parameters: sparsity $\tau = 39$, range of secret $\eta = 2$									
3	L	256	SKA	HPC3	4535	n/s*	79872	274	n/s*
	A	1	RCA	HPC3	10	45	216	10375	19.4
	R/ALRP	1	RCA	HPC2	15	51	108	10726	20.0
	ALP	2	RCA	HPC3	14	87	432	5383	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	—	12594	8.7
	ALP	16	SKA	—	2	216	—	834	0.7
1	L	256	SKA	HPC3	1192	n/s*	14336	344	n/s*
	A	1	RCA	HPC3	4	30	40	13084	16.7
	R/ALRP	1	RCA	HPC2	6	32	20	13574	17.3
	ALP	2	RCA	HPC3	4	55	80	6812	13.6
2	L	256	SKA	HPC3	2744	n/s*	43008	344	n/s*
	A	1	RCA	HPC3	7	40	120	13084	24.3
	R/ALRP	1	RCA	HPC2	11	45	60	13574	25.2
	ALP	2	RCA	HPC3	9	76	240	6812	13.8
3	L	256	SKA	HPC3	4931	n/s*	86016	344	n/s*
	A	1	RCA	HPC3	11	51	240	13084	24.3
	R/ALRP	1	RCA	HPC2	18	58	120	13574	25.2
	ALP	2	RCA	HPC3	16	99	480	6812	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	—	15421	10.5
	ALP	16	SKA	—	2	194	—	1021	0.7
1	L	256	SKA	HPC3	1096	n/s*	13312	421	n/s*
	A	1	RCA	HPC3	3	26	36	15961	25.7
	R/ALRP	1	RCA	HPC2	5	28	18	16501	26.6
	ALP	2	RCA	HPC3	4	49	72	8281	16.7
2	L	256	SKA	HPC3	2522	n/s*	39936	421	n/s*
	A	1	RCA	HPC3	6	36	108	15961	29.7
	R/ALRP	1	RCA	HPC2	10	39	54	16501	30.7
	ALP	2	RCA	HPC3	8	68	216	8281	16.7
3	L	256	SKA	HPC3	4534	n/s*	79872	421	n/s*
	A	1	RCA	HPC3	10	45	216	15961	29.7
	R/ALRP	1	RCA	HPC2	15	51	108	16501	30.7
	ALP	2	RCA	HPC3	14	88	432	8281	16.8

*not synthesizeable within 72 hours

Table 18. Additional DSE and synthesis results for Kyber-CPA decryption, cf. Table 8

d	Opt.	Design Config.				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*	—	4102	n/s*
	A	1	1	sRCA	—	42	120	—	7276421	11.42
	ALP	16	43	SKA	—	85	1180	—	7957	0.05
1	L	256	256	SKA	HPC3	6201	n/s*	86016	14857	n/s*
	A	1	1	sRCA	HPC3	174	171	156	13174792	21.89
	R	1	1	sRCA	HPC2	178	174	78	19073035	34.12
	ALRP	1	1	pRCA	HPC2	191	205	126	244747	0.34
	ALP	3	10	SKA	HPC3	398	518	3304	34142	0.07
2	L	256	256	SKA	HPC3	14314	n/s*	258048	14857	n/s*
	A	1	1	sRCA	HPC3	392	225	468	13174792	25.68
	R	1	1	sRCA	HPC2	401	232	234	19073035	36.82
	ALRP	1	1	pRCA	HPC2	431	277	378	244747	0.44
	ALP	3	10	SKA	HPC3	908	821	9912	34142	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	—	9702	14.2
	ALP	16	43	SKA	—	85	1180	—	11	0.1
1	L	256	256	SKA	HPC3	6201	n/s*	86016	20	n/s*
	A	1	1	sRCA	HPC3	174	171	156	17566	31.8
	R	1	1	sRCA	HPC2	178	174	78	25431	42.5
	ALRP	1	1	pRCA	HPC2	191	205	126	326	0.5
	ALP	3	8	SKA	HPC3	398	360	2648	45	0.1
2	L	256	256	SKA	HPC3	14314	n/s*	258048	20	n/s*
	A	1	1	sRCA	HPC3	392	225	468	17566	37.9
	R	1	1	sRCA	HPC2	401	232	234	25431	49.4
	ALRP	1	1	pRCA	HPC2	431	277	378	326	0.5
	ALP	3	10	SKA	HPC3	908	821	9912	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	—	9964	17.6
	ALP	16	32	SKA	—	93	213	—	14	0.1
1	L	256	256	SKA	HPC3	7738	n/s*	104448	22	n/s*
	A	1	1	sRCA	HPC3	180	175	228	18091	32.0
	R	1	1	sRCA	HPC2	186	181	114	26217	46.4
	ALRP	1	1	pRCA	HPC2	200	212	162	329	0.6
	ALP	2	8	SKA	HPC3	399	381	3216	54	0.1
2	L	256	256	SKA	HPC3	17836	n/s*	313344	22	n/s*
	A	1	1	sRCA	HPC3	404	234	684	18091	39.8
	R	1	1	sRCA	HPC2	420	245	342	26217	57.4
	ALRP	1	1	pRCA	HPC2	452	290	486	329	0.6
	ALP	2	8	SKA	HPC3	910	623	9648	54	0.1

*not synthesizeable within 72 hours

Table 19. Additional DSE and synthesis results for Kyber-CCA decapsulation, cf. Table 9.

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*	—	19612	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	75	316	—	19960068	75321
	ALP	10	SKA	SKA	pRCA	100	—	91	459	—	72036	541
1	L	256	SKA	SKA	SKA	1600	HPC3	7869	n/s*	105922	69220	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	259	447	326	36368922	136213
	R	1	sRCA	sRCA	sRCA	25	HPC2	268	454	163	52707120	205887
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	282	485	211	931632	3741
	ALP	8	SKA	SKA	pRCA	200	HPC3	505	668	3880	138441	541
2	L	256	SKA	SKA	SKA	1600	HPC3	18095	n/s*	317766	69220	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	544	575	978	36368922	137241
	R	1	sRCA	sRCA	sRCA	25	HPC2	564	590	489	52707120	198895
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	596	635	633	931632	3516
	ALP	8	SKA	SKA	pRCA	200	HPC3	1113	1013	11640	138441	543
		Algorithm parameter: Kyber-768 ($k = 3, \eta = 2$)										
0	L/ALP	256	SKA	SKA	SKA	1600	—	343	n/s*	—	28170	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	74	311	—	36416346	142251
1	L	256	SKA	SKA	SKA	1600	HPC3	6333	n/s*	87486	107005	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	254	433	250	66103166	250391
	R	1	sRCA	sRCA	sRCA	25	HPC2	260	439	125	95698594	359769
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	273	469	173	1557154	6254
	ALP	10	SKA	SKA	pRCA	200	HPC3	504	797	3956	209081	791
2	L	256	SKA	SKA	SKA	1600	HPC3	14574	n/s*	262458	107005	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	532	553	750	66103166	265475
	R	1	sRCA	sRCA	sRCA	25	HPC2	546	564	375	95698594	359769
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	576	609	519	1557154	6155
	ALP	10	SKA	SKA	pRCA	200	HPC3	1179	1179	11868	209081	795
		Algorithm parameter: Kyber-1024 ($k = 4, \eta = 2$)										
0	L/ALP	256	SKA	SKA	SKA	1600	—	343	n/s*	—	42122	n/s*
	A	1	sRCA	sRCA	sRCA	25	—	74	312	—	58260482	218204
1	L	256	SKA	SKA	SKA	1600	HPC3	6333	n/s*	87486	159596	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	254	433	250	105694008	395858
	R	1	sRCA	sRCA	sRCA	25	HPC2	260	439	125	153015406	577417
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	273	469	173	2389102	9595
	ALP	10	SKA	SKA	pRCA	200	HPC3	504	797	3956	322303	1259
2	L	256	SKA	SKA	SKA	1600	HPC3	14574	n/s*	262458	159596	n/s*
	A	1	sRCA	sRCA	sRCA	25	HPC3	532	553	750	105694008	424474
	R	1	sRCA	sRCA	sRCA	25	HPC2	546	564	375	153015406	614520
	ALRP	1	pRCA	sRCA	sRCA	25	HPC2	576	609	519	2389102	9595
	ALP	10	SKA	SKA	pRCA	200	HPC3	1110	1179	11868	322303	1207

*not synthesizeable within 72 hours

C Masked Keccak Comparison

A comparison of our Keccak designs with related work [BDN$^+$13, SM21, GSM17] is shown in Table 20. It should be noted that our designs can hardly compete with the handcrafted implementations because several low-level optimizations are not yet available with our tool.

D Comparison to Streamlined NTRU Prime

Land et *al.* [LMRG24] recently presented a gadget-based implementation of Streamlined NTRU Prime. Table 21 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 20. 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 10).

Ref.	Opt.	d	Area	Rand.	Lat.	Delay	Technology
			[kGE]	[bit]	[cycles]	[ns]	
[BDN+13]		1	116.6	4	25	42.2	NanGate 45 nm
[BDN+13]		1	39.0	4	1625	2561.2	NanGate 45 nm
[GSM17]		1	18.7	0	3160	3690.7	UMC 130 µm
[GSM17]		1	22.3	0	1648	2028.8	UMC 130 µm
[GSM17]		1	108.0	0	25	28.2	UMC 130 µm
[SM21]		1	129.3	0	72	92.9	UMC 130 µm
HADES	L	1	149.4	3200	120	40.8	NanGate 45 nm
HADES	A	1	30.1	50	3144	1792.5	NanGate 45 nm
HADES	R/ALRP	1	34.6	25	4680	3744.0	NanGate 45 nm
HADES	ALP	1	61.6	800	264	198.1	NanGate 45 nm
[GSM17]		2	28.8	75	3160	3706.7	UMC 130 µm
[GSM17]		2	34.6	75	1648	1951.2	UMC 130 µm
[GSM17]		2	232.3	4800	25	29.8	UMC 130 µm
[SM21]		2	231.5	0	72	108.0	UMC 130 µm
HADES	L	2	325.9	9600	120	40.8	NanGate 45 nm
HADES	A	2	52.2	150	3144	2954.9	NanGate 45 nm
HADES	R/ALRP	2	54.1	75	4680	2901.4	NanGate 45 nm
HADES	ALP	2	117.7	2400	264	211.2	NanGate 45 nm

Table 21. 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 similar security compared to sNTRUp-761 (NIST level 2).

d	Scheme	Area	SRAM	Rand.	Latency	Delay
		[kGE]	[bit]	[bit]	[cycles]	[μs]
1	sNTRUp-761	201	189440	310	1870049	9034
	Kyber-512	447	28608	326	36368922	136213
	Kyber-512	454	28608	163	52707120	205887
	Kyber-512	485	28608	211	931632	3741
	Kyber-512	668	28608	3880	138441	541
	Kyber-768	433	35776	250	66103166	250391
	Kyber-768	439	35776	125	95698594	359769
	Kyber-768	469	35776	173	1557154	6254
	Kyber-768	797	35776	3956	209081	791
2	sNTRUp-761	373	246272	930	1870049	11334
	Kyber-512	575	30400	978	36368922	137241
	Kyber-512	590	30400	489	52707120	198895
	Kyber-512	635	30400	633	931632	3516
	Kyber-512	1013	30400	11640	138441	543
	Kyber-768	553	38336	750	66103166	265475
	Kyber-768	564	38336	375	95698594	359769
	Kyber-768	609	38336	519	1557154	6155
	Kyber-768	1179	38336	11868	209081	795