IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–34. DOI:XXXXXXXX

TETRIS: Automated Design Space Exploration of Randomness-Latency Trade-offs in Masked Hardware

Nilotpola Sarma[1], Tapish Patidar[1], Nupur Brahamanya[1] and Chandan Karfa[1]

Indian Institute of Technology Guwahati, Guwahati, India
s.nilotpola@iitg.ac.in, p.tapish@iitg.ac.in, b.nupur@iitg.ac.in, ckarfa@iitg.ac.in

Abstract.
Given a fixed security order, the randomness and latency of masked hardware present a trade-off. This trade-off has not been structurally examined well enough to enable an efficient search for a user-optimal (randomness/latency-efficient) masked design. Gadget-based masking simplifies masking by replacing each unmasked primitive circuit with a corresponding masked circuit, called a *gadget*, to realize a larger masked design. These gadgets, in turn, have masking-order-dependent latency-randomness costs, lending a structure to the randomness and latency of gadget-based masked hardware. This structure enables automated Design-Space Exploration (DSE) of gadget-based masked hardware that takes in a user's constraints on randomness (or latency) to arrive at the latency (or randomness) optimal assignment of gadgets with less area and vice versa. This article introduces a software-level DSE approach, the basis of which is the two DSE algorithms - Minimize Latency under Randomness Constraints (MLRC) and Minimize Randomness under Latency Constraints (MRLC), which are duals of each other. While prior work optimizes masked hardware by formulating a global optimization based on the Boolean Satisfiability Problem (SAT), our results show that gadget-based masked hardware using Probe-Isolating Non-Interference (PINI) gadgets embodies a structured trade-off, enabling efficient heuristic-based solutions rather than relying on heavy global optimizations. This gives our tool comparable or superior area results in under a millisecond - a speedup of up to several orders of magnitude to the State-of-the-Art tools.

Keywords: Power Side Channel Security · Masking · Low-latency masking · Low-randomness masking · Design Space Exploration · High-Level Synthesis

1 Introduction

Power Side-Channel Attacks (PSCAs), first introduced by Kocher et al. [KJJ99] continue to be one of the significant threats to the security of embedded systems. Due to the physical characteristics of computing hardware, it is possible to correlate the secret keys of mathematically secure cryptographic algorithms to the overall power consumption of the device running them. Specifically, all intermediate computation results of an algorithm can be correlated to the power consumption of the host device. An intelligent attacker can passively tap into the power consumption information and effectively exploit this correlation to reveal the secrets at a very low cost. Countermeasures against PSCAs are therefore a requirement in modern secure-system design. Masking [CJRR99] is one of the popular countermeasures against such attacks that splits the secret inputs into random shares and processes them independently via a split data-path, making sure to re-randomize any insecure combination of shares. This causes all intermediate computation,

Licensed under Creative Commons License CC-BY 4.0.

and consequently the overall power consumption, to be randomized and hence decorrelated from the secrets.

Masking is an intricate process, requiring a vast skill set encompassing design, verification, and cryptography. In addition, masking incurs high overheads in terms of latency and randomness generation. Masking of computations like the bit-wise AND operation requires additional circuitry to deal with possible re-combination of shares. To secure these recombinations, further randomness is incorporated into these computations to avoid re-combination of shares via logic, and extra register stages are added to avoid re-combination of shares due to glitches. The usage of the optimal set of these extra circuit elements for masking an overall design consisting of several computations is not straightforward. One way of masking cryptographic hardware relies on a modular approach called *gadget-based* masking [GMK16, CGLS20, KM23]. This encapsulates the masked versions of common affine and non-affine circuit elements (like XOR and AND operations, respectively) into *gadgets* (like the AND gadgets in [CGLS20]) such that they can be trivially *composed* to form the overall masked design. This vastly simplifies the design process.

However, there are several *gadgets* that can be used to mask a cryptographic hardware, each with a specific latency and randomness cost. *Added to this is the general observation for masked hardware design that an increase in latency can be compensated for a reduced number of bits of randomness-generation and vice-versa, presenting a trade-off to the designer.* Traditionally, only one type of gadget would be used to mask all non-linear operations in an unmasked design to arrive at a gadget-based masked design [CGLS20, KMMS21]. Recent approaches have shown that if *composability rules* are followed, several gadgets can be used to replace the different non-linear operations in a design to arrive at a masked design with a reduced area benefit [CGM+24]. Consequently, there are several combinations of gadgets in a masked design that can result in incurring a range of overall latency and randomness costs. *Thus, arriving at the optimal masked design based on a user's randomness-latency budget constitutes solving a Design-Space Exploration (DSE) problem.*

A manual DSE of gadget-based masked hardware for large cryptographic benchmarks seems impractical. Few recent works [KMMS21, CGM+24] target automatic masked hardware design at the gate level. However, such approaches either use the same gadgets to mask all gates in the same design [KMMS21] or use heavy global optimization strategies to solve constraints and arrive at an optimal masked design as in COMPRESS [CGM+24]. In addition, they do not look at the dual nature of masked hardware optimization, which is that increasing randomness facilitates a decrease in latency and vice versa. *Such formulations implicitly assume limited structure in the underlying design space, leading to high computational complexity.* We demonstrate that gadget-based masked hardware contains enough structure to allow for a more efficient, heuristic-based exploration of the design space. Recent works have also suggested the High-Level Synthesis (HLS) of masked hardware directly from masked software [SSM21, STK24], enabling automated masked hardware generation from an algorithm's unmasked software implementation, making a high-level design space exploration tool necessary.

Motivated by this, we propose TETRIS - a software-level DSE of masked hardware that incorporates two DSE approaches, MLRC and MRLC, that facilitate the DSE of masked hardware based on a user's target randomness and target latency, respectively. This DSE flow can be integrated with a secure HLS flow [STK24] to generate a user-optimal power side-channel secure masked hardware from unmasked software in quick time. The contributions of this work are as follows:

- This work presents a DSE flow TETRIS with two DSE algorithms, Minimize randomness under a latency constraint (MRLC) and minimize latency under a randomness constraint (MLRC), that are duals of each other, for a systematic exploration of the latency-randomness trade-off in gadget-based masked hardware at the behavioral level. To the best of our knowledge, no prior work explicitly formulates or solves the

problem of MLRC in masked hardware design.

- TETRIS is equipped to handle Toffoli gadget candidates and the asymmetric natures of the latency of a few gadgets.
- TETRIS has been integrated into a secure HLS flow [STK24] to obtain masked hardware from the behaviorally optimized masked software obtained via the DSE flow presented.
- TETRIS took less than a millisecond for the DSE of the masked hardware of all three Sbox benchmarks (AES Canright's, AES Boyar and Peralta's, and SKINNY Sboxes). The randomness achieved by our tool is always equal to or better than the SOTA COMPRESS tool [CGM+24] (with the COMAR gadget), and the area results are comparable to COMPRESS. In addition, we obtained an impressive speedup (in times) of: Boyar and Peralta S-box: ≈ 78.75 times, Skinny S-box: ≈ 28.18 times, Canright S-box: ≈ 1.43 times on average over corresponding designs generated via COMPRESS using our MRLC algorithm.
- The results over three benchmark designs, four security orders of masking and a gadget library of at least 5 gadgets, including gadgets with asymmetric latency, demonstrate that design space exploration of masked hardware may highly benefit from local heuristics than global constraint solving.

The rest of the paper is organized as follows. Section 2 presents the related works. Section 3 presents the background on gadgets and gadget-based masking. Section 4 provides the motivation behind the user-defined constraint-based DSE of masked designs done by TETRIS. Section 5 presents the overall TETRIS tool flow. Section 6 and Section 7 present the DSE algorithms: MLRC and MRLC, used in TETRIS, respectively. Section 8 presents the complexity of the algorithms. Section 9 presents the heuristics that can be used to exploit the asymmetry of gadgets to realize optimized masked designs with MLRC and MRLC. Section 10 discusses the inclusion of the COMAR gadget in our DSE algorithms. Section 11 provides a brief overview of pipeline register optimization in masked designs. Section 12 presents a combined flow using the two algorithms to realize a design with both latency and randomness constraints as a use case. Section 13 presents the experimental evaluation of TETRIS on the DSE of four benchmark cryptographic designs. Section 14 concludes the article.

2 Related Works

Hardware masking is costly in terms of randomness generation and latency. Each year, several works are published at elite venues in cryptography and hardware security that introduce new and efficient techniques to optimize masked hardware based on these two parameters. These optimization techniques can be discussed in the following sub-sections:

2.1 Masking-scheme/gadget-independent techniques

Optimization of masked circuits began as early as the first masking scheme [CJRR99]. From the onset, it was clear that masking schemes, though formally sound, had significant randomness and area costs. Several masking schemes [CJRR99, CG00] with similar overheads were proposed under the Ishai-Sahai-Wagner (ISW) masking scheme [ISW03]. However, with the realization that glitches in hardware alter masking security [MPG05], and the scalability issues of Threshold Implementations [BGN+14], new masking schemes were introduced that suggested insertion of registers at specific locations to arrest glitches and conserve masking security in a glitchy circuit [GMK16]. This introduced latency costs in masked hardware design. Generic optimization techniques for reducing latency and area that are generic to digital design [Mic94] began to be tailored to masking schemes

[CGM+24, STK24]. As straightforward as it might seem, applying generic optimizations to masked designs might not result in secure masked hardware [SSM21, IYL+23, STK24].

2.2 Gadget-based optimization techniques

With the advent of new masking schemes, their verifiability became challenging. Specifically, hardware masking verification algorithms were found to fail to verify higher-order masked designs [MMSS19]. Thus, there was a need for secured-by-construction masking schemes that could be verified incrementally by composing sub-circuits that are already verified [BBD+16, CS20]. This led to the introduction of subcircuits called *gadgets*, strategically chosen to allow tunable overhead and to enforce strict masking guarantees that can yield an overall masked design. The introduction of gadgets led to a series of works, each suggesting a new (set of) gadgets that can achieve lower randomness or latency than the State-of-the-art, some promising both, albeit with more restrictive compositional scenarios. This led to a serious need to optimize gadget-based masked designs. With their seminal work [KMMS21], Knichel et al. initiated a series of works [KM22, KSM22] in this regard. Thereafter several gadget-specific randomness-reuse techniques e.g., [HB24] for HPC3.1 and HPC1 enabling a better masking of Canright's original F_{2^8} inverter. TETRIS is a general DSE framework that makes no assumptions about the underlying algebraic structure.

2.3 Design-dependent optimization techniques

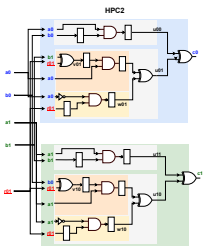
With the advent of new gadgets and the gadget-based masking technique, it soon became evident that gadgets, while enabling seamless composition and verification, offered several opportunities for optimization. The same gadget, depending on its design, can be modified across its instances, resulting in reduced overall randomness or latency without any degradation in security. An example of this is the Smooth Passage with Guards [GKM+24] that modifies the Domain-Oriented Masking (DOM) AND gadget [GMK16] into two other types with lower overhead and strategically uses them in Canright's AES S-box, with lower overall randomness usage. These techniques, however, are design- and masking-scheme-specific, making their generic application and exploration difficult.

2.4 Automated design-space exploration and synthesis of masked designs

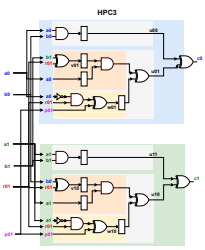
AGEMA [KMMS21] presents an automated tool for generating masked cryptographic hardware by applying transformation techniques and different masking schemes to achieve an optimal design. Their tool, however, takes as input a gate-level Verilog design on which the transformations are performed. However, they do not explicitly find a minimum-latency masked design under a randomness constraint, or a minimum-randomness masked design under a latency constraint. In that aspect, TETRIS provides that choice to the user. Also, TETRIS performs design-space exploration at the C level rather than in Verilog, making it more efficient. COMPRESS [CGM+24] formulates the minimization of randomness and area for a specific latency requirement on a masked design as a global constraint-solving optimization problem over gadget selection, scheduling, sub-computation re-use, and register placement. While effective, this approach treats the design space as largely unstructured, leading to significant computational overhead. In contrast, TETRIS observes that the randomness-latency tradeoff presents an underlying structure to the design space exploration, which, if exploited, could lead to faster heuristic optimization with comparable overhead (area/randomness/latency). In addition, none of the previous works have targeted the MLRC problem.

3 Background

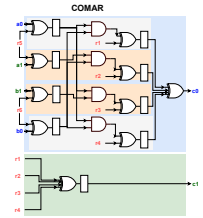
In this section, we present a brief background of the material required to understand the content of the paper.



(a) 1st-order HPC2 gadget



(b) 1st-order HPC3 gadget



(c) 1st-order COMAR gadget

```
void hpc2(bool a0, bool a1, bool
       b0, bool b1, bool *y0,
      bool *y1, bool r0){
    bool u0, u1, i0, i1, v0, v1,
           p0, p1, p01, p2, p3,
          p23, a0_neg, a1_neg;
    a0_neg = !a0;
    a1_neg = !a1;
    w01 = a0_neg & reg(r01);
    w10 = a1_neg & reg(r01);
    v01 = b1 ^ r01;
    v10 = b0 ^ r01;
    u00 = a0 & reg(b0);
    u01_ = a0 & reg(v01);
    u01 = reg(u01_) ^ reg(w01);
    *y0 = reg(u00) ^ u01;
    u11 = a1 & reg(b1);
    u10_ = a1 & reg(v10);
    u10 = reg(w10) ^ reg(u10_);
    *y1 = reg(u11) ^ u10;}
```

Listing (1) HPC2 Implementation

Figure 1: Gadgets for masking against Power Side-Channel Attacks

Gadget-based masking: Gadgets are small circuits that compute basic *non-linear* operations, such as bitwise multiplication, which can be composed using *linear* operations, such as XOR, to yield a masked PSCA-secure overall circuit. Several gadgets for masked multiplication, like DOM [GMK16], HPC1 [CGLS20], HPC2 [CGLS20], HPC3 [KM22], and COMAR [KM23], exist that map a set of shared inputs to a set of shared outputs, allowing for seamless composition to obtain an overall shared design. A class of *non-linear* gadgets, HPC, was introduced in [CGLS20]. The strategy was to *re-mask* the inputs to the gadgets before multiplying. In HPC2, all the inputs that have been split into shares of two are registered. The computations are performed as follows: $c0 = ((a0 \otimes r01) \oplus (b1 \otimes r01)) \oplus (a0 \otimes b0)$ and $c1 = ((a1 \otimes r01) \oplus (b0 \otimes r01)) \oplus (a1 \otimes b1)$ with registers being placed at all the input shares and four intermediate locations. The HPC2 multiplication gadget using two shares is shown in Figure 1a and Listing 1. The HPC3 [KM22] gadget for two shares is shown in Figure 1b. It performs a similar computation of the product terms but with a different placement of re-masking gates and registers, requiring an extra random bit. The two-share HPC3 gadget incurs a latency of 1 cycle and needs two bits of fresh randomness. There also exist masking gadgets that perform a combination of operations. An example of such gadgets are the Toffoli gadgets (HPC2o, HPC3o) that perform a masked AND-XOR operation [CGM+24] secure a HPC2 or HPC3o masked AND with a masked XOR. These gadgets have the same latency and randomness overhead as the original HPC2 or HPC3 gadgets, but lead to better area optimization. In this work, we focus on the DSE of masked hardware using only PINI gadgets.

4 Motivation

This section aims to demonstrate the trade-off between randomness and latency in masked hardware. It then demonstrates the need for an automated Design Space Exploration (DSE) of the latency-randomness trade-off in masked hardware. It also points out the striking linear correlation between the area of masked gadgets and the number of balancing (pipeline) registers used to mask them.

The Randomness-Latency Trade-Off: For the 1st-order gadgets presented in Figure 1, we can see that their randomness and latency vary. Table in Figure 2 presents the randomness cost (number of random bits needed for re-masking) and the latency incurred by each input (port) of these gadgets in the $1^{st}, 2^{nd}$, 3rd and 4th masking order. We can observe that the 1st-order HPC1 gadget incurs a latency of 2 and a randomness cost of 2 bits. The HPC2 gadget incurs the same latency but at reduced randomness. Similarly, HPC3 has lower latency but higher randomness. Similar trade-offs can be observed for

(a) Comparison of gadget implementations across masking orders.

d	Gadget	Lat.(in1)	Lat.(in2)	Rand.	Area (wo rand)	#regs
2	hpc1	2	1	2	50.7	6
2	hpc2o	2	1	1	60.64	10
2	hpc3o	1	1	2	43.09	6
2	hpc2	2	1	1	79.26	14
2	hpc3	1	1	2	55.32	8
3	hpc1	2	1	5	127	12
3	hpc2o	2	1	3	163.85	27
3	hpc3o	1	1	6	115.71	15
3	hpc2	2	1	3	193.9	33
3	hpc3	1	1	6	134.06	18
4	hpc1	2	1	10	217.3	20
4	hpc2o	2	1	6	315.47	52
4	hpc3o	1	1	12	222.37	28
4	hpc2	2	1	6	358.56	60
4	hpc3	1	1	12	246.84	32
5	hpc1	2	1	15	325	30
5	hpc2o	2	1	10	515.50	85
5	hpc3o	1	1	20	363.09	45
5	hpc2	2	1	10	573.23	95
5	hpc3	1	1	20	393.68	50

Area (wo rand): Area using Yosys 0.33 + Nangate 45nm without randomness generation area.
#regs indicates register count from TETRIS.

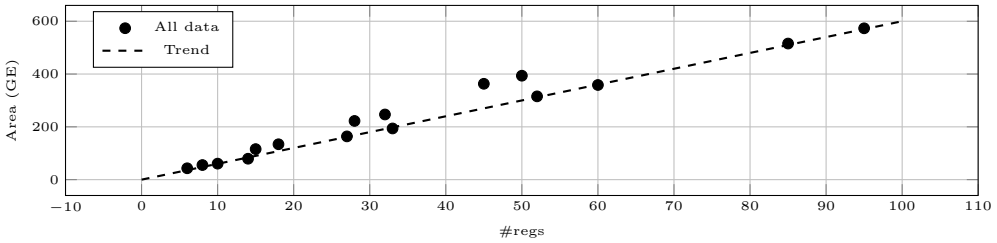


Figure 2: Area vs Register Count across gadgets and masking orders.

higher orders, demonstrating the randomness-latency trade-off in masked hardware.

Asymmetry of Gadgets: Another aspect is the asymmetric nature of the latency of these gadgets, where the latency of all paths in the gadgets is not the same. For example, the latency of the path from input 1 to the output of hpc2 is 2, whereas the same path from input 2 is 1. This poses another unique challenge in the DSE. An additional challenge is incorporating the Toffoli gadgets into the DSE to further optimize area.

Pipeline Register vs Area Overhead for Masked Hardware: It is observed that the area of a masked design increases linearly with the number of pipeline registers. Consider the scatter plot in Figure 2, where each point in the plot corresponds to a specific gadget of a specific order. The scatter plot highlights a strong positive correlation between register count and area, independent of gadget type or masking order. This indicates that the reduction of pipeline registers is crucial for optimal masking.

5 TETRIS: The Overall Flow

Gadget-based masked hardware generation using our DSE flow, TETRIS, is a five-step process as discussed below. The overall flow of TETRIS is given in Figure 3.

5.1 Step 1: Parse to AST and Identify Toffoli gadget [CGM+24] candidates

The input C code is assumed to contain no loops or conditional (if-else) constructs, as in the SOTA DSE works on masking [CGM+24]. The initial step parses the behavioral specification of the unmasked design in C and constructs its Abstract Syntax Tree (AST). At this stage, we use the techniques in [CGM+24] to extract AND-XOR gates such that the output of the AND gate is used as input by one and only XOR gate. These are marked

as *Toffoli gadget candidates*. Apart from the XOR gates, which are a part of a Toffoli gadget candidate, all other XOR gates, along with NOT gates, are extracted as *linear gates*. As given in Lemma 1, the best strategy from a DSE perspective is to find as many Toffoli gates as possible and mark them as *Toffoli gate candidates* to be assigned to a Toffoli gate from the *gadget library* (Table in Figure 2) during the DSE (MLRC/MRLC). These are marked in the AST for actual assignment in Step 3.

Lemma 1. *Given an unmasked AST with only XOR, AND, and NOT operations, a gadget-based masked design using the maximum possible Toffoli gates will always correspond to the design with the least randomness under a latency constraint with the minimum area [CGM+24].*

Proof. Can be derived from [CGM+24]. □

5.2 Step 2: Identify *isolated* XOR and other *linear gadget* assignments

After identifying the *Toffoli gadget candidates*, we find the rest of the XOR operations that are not coupled with any AND operation. These can be replaced by their masked variants, one of which is shown in Algorithm 1. The rest of the *linear* operations (which are the NOT operation in our case) are assigned their masked variants as in Algorithms 1 and 2. For each input design, all the gadget assignments are stored in a tabular structure called *gadget assignment*.

A note on Linear Gadgets in DSE: Note that, in some gadget-based masking scenarios with non-linear and linear gadgets composed in the same design, specifically in NI/SNI gadget-based masking, there may be an opportunity for including several linear gadgets in the DSE and performing a DSE with linear gadgets as well, leading to reduced randomness. The root of such optimizations is the need to refresh the NI/SNI non-linear gadgets when they are composed. However, as established in the existing literature [CS19], **linear gadgets do not require refresh for compositions with PINI non-linear gadgets**, and TETRIS only considers PINI non-linear gadgets. The randomness optimizations due to linear gadgets and their compositions with non-linear gadgets do not apply when the non-linear gadgets are PINI. Thus, when non-linear gadgets are PINI, linear operations are share-by-share with no refresh (Algorithms 1 and 2), ensuring trivial composition. The benefits lost from using PINI gadgets are also quite less. For low-to-moderate masking orders ($d \leq 20$, which is practical in most cases), PINI composition is randomness-competitive with optimized NI/SNI composition [CS19].

5.3 Step 3: Gadget-tree construction

At this point, since the gadget assignment of the linear gadgets are done away with in Step 2, and a gadget-tree with only the non-linear gadgets (including the Toffoli gadget candidates) is abstracted out to construct a *gadget-tree* of the design. In contrast to the AST, where the fan-in of a node corresponds to an intermediate computation result in the 3-address code of the design, the fan-in of a node in the gadget-tree does not reflect the actual input fan-in of an AND operation (or a 3-input Toffoli gate) in Step 1. A gadget tree for the design in Figure 3(a) is shown in Figure 3(c).

5.4 Step 4: MLRC or MRLC

The gadget tree provides an abstraction that aids the DSE of gadget-based masked hardware. The DSE of masked hardware with a randomness-latency trade-off is performed using the two algorithms, MLRC and MRLC, according to the user's choice. The user can specify either *target_randomness* or *target_latency*, and the assignment of gadgets must strictly meet the corresponding requirement. The DSE output is an optimal gadget assignment represented as a tabular structure that assigns a corresponding gadget to each non-linear operation (AND) or Toffoli gadget candidate (identified in Step 1). An example

of such an assignment is shown in Figure 3(d). The optimal assignments of gadgets to the gadget-tree are used to obtain a gadget-based masked AST of the implementation as the output of this step. To accommodate the benefits of asymmetric gadgets and their efficient *packing*, reducing pipeline registers and randomness, the DSE algorithms are supplemented by heuristics at this stage, details of which are given in Section 9.

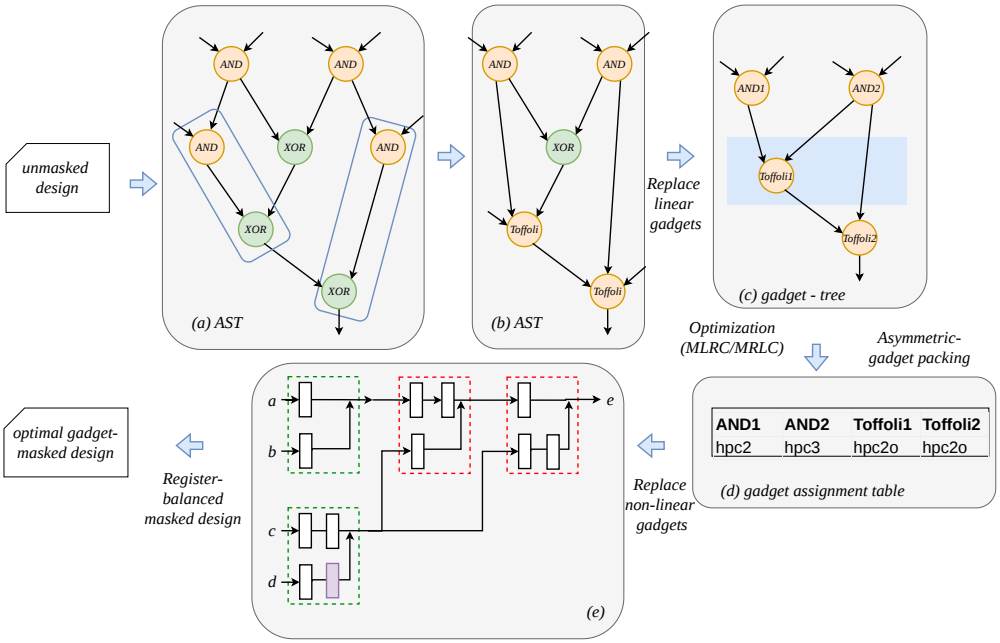


Figure 3: Steps of TETRIS DSE Flow: (a) Step 1: Parse unmasked design and extract AST, identify Toffoli gate replacement candidates, (b)Step 2: Identify isolated XOR and other linear gadget-replacement candidates: replace them with available PINI gadgets, (c) Step 3: Extract *gadget-tree* from the remaining gadgets and their dependencies, (d) Step 4: Run our optimization Algorithms according to the user's requirements: obtain a gadget assignment, followed by optimization with asymmetric gadgets (e) Step 5: Synthesis to RTL

Algorithm 1: Linear Gadget with d shares [CGM+24]

Input: Shares $\mathbf{x} = (x_0, \ldots, x_{d-1})$, $\mathbf{y} = (y_0, \ldots, y_{d-1})$, linear gate X
Output: Shares $\mathbf{z} = (z_0, \ldots, z_{d-1})$
for $i \leftarrow 0$ **to** $d-1$ **do**
  $z_i \leftarrow X(x_i, y_i)$;
end
return $\mathbf{z}$;

Algorithm 2: Linear NOT Gadget with d shares [CGM+24]

Input: Sharing $\mathbf{x} = (x_0, \ldots, x_{d-1})$
Output: Sharing $\mathbf{z} = (z_0, \ldots, z_{d-1})$
$z_0 \leftarrow \neg x_0$;
for $i \leftarrow 1$ **to** $d-1$ **do**
  $z_i \leftarrow x_i$;
end
return $\mathbf{z}$;

5.5 Step5: Register Balancing and Synthesis to RTL

After the optimized gadget-based masked C code is generated in the above step, the gadget-based masked hardware design can be generated via HLS from that code. In this work, we adopted the masking-specific HLS tool MaskedHLS [STK24], which ensures that PSCA security is maintained throughout HLS. MaskedHLS generates the masked Register-Transfer Level (RTL) with a reduced number of registers, ensuring minimum

latency possible with the given datapath. MaskedHLS ensures the latency determined by TETRIS is maintained after HLS, since it identifies the minimum possible pipeline stages in the circuit that satisfy all necessary register requirements imposed by the gadgets. The output of MaskedHLS is the gadget-based masked RTL code of the design.

In this work, we present two DSE algorithms, MLRC and MRLC, for systematically exploring the latency-randomness trade-off in gadget-based masked hardware. The following sections provide a detailed discussion of these algorithms.

A note on Area (GC): It must be noted that finding the actual area after synthesis (using Synopsys Design Compiler or Yosys) for each gadget assignment to minimize it during DSE is time-consuming. Hence, to aid the DSE process, we will rely on a simple heuristic we have named *Gate Count (GC)*. We synthesized the designs for each gadget via Yosys using the Nangate 45nm Open Cell Library. We found that the area of the gadgets in the gadgets list roughly follows a ratio of $HPC3 : HPC2 \approx 3 : 4$ across all orders. Thus, we defined Gate Count (GC) as follows:

$$GC = (no_of_HPC3 \times 3) + (no_of_HPC2 \times 4)$$

For designs with COMAR, this becomes:

$$GC = (no_of_HPC3 \times 3) + (no_of_HPC2 \times 4) + (no_of_COMAR \times 5)$$

Similarly, GC values can be obtained using Toffoli gadgets HPC3o and HPC2o. Note that this ratio might vary for higher-order gadgets; hence, a presynthesis of the individual gadgets in that order is required. TETRIS will work seamlessly with any other heuristic for the area that the user provides.

6 Minimize Randomness under Latency Constraint (MRLC)

After Step 4 in Section 5, a *gadget_tree* is obtained, with placeholders for placing Toffoli gadgets or AND gadgets. Given a *Gadget_tree* with L AND levels, and a list of gadgets in *Gadgets_list* at masking order d, each gadget g is characterized by its latency $g.latency$ and randomness $g.rand$. For a user-specified $target_latency$, MRLC computes a gadget assignment that minimizes randomness while satisfying this latency constraint. While making a gadget assignment, MRLC will check if the *gadget_tree* node is a Toffoli gadget replacement candidate as marked in Step 1 in Section 5, if so, it will replace it with a Toffoli gadget as HPC2o or HPC3o.

6.1 MRLC: Greedy Algorithm ($d \geq 2$)

The objective of the MRLC algorithm is to obtain a design under the user-provided $target_latency$ constraint. Once this is met, the algorithm tries to optimize the randomness cost: the number of bits of fresh randomness required for masking each gadget in the design. Thus, the algorithm should start with the least-latency gadgets assigned to the *gadget_tree* and then iteratively use the remaining latency ($buffer_latency$) at each step to revise the gadget assignment for each *gadget_tree* node with higher latency, assigning a lower-randomness gadget to reduce overall randomness cost. Thus, MRLC has two steps: an initial assignment of gadgets and a reduction of randomness using $buffer_latency$.

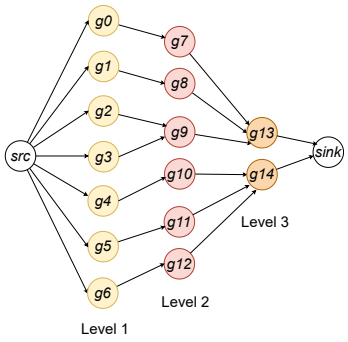


Figure 4: An example design Gadget-tree to illustrate MRLC

Algorithm 3: `ReduceRandomnesswithBufferLatency`

Input: $gadget_tree, mrlc_levels, buffer_latency, d$
Output: $gadget_tree$ with optimized latency allocation across levels

```
for each level in mrlc_levels do
    if buffer_latency ≥ 1 then
        for each gadget in level do
            // replace gadget with
               low-randomness/higher-latency (by 1)
            mrlc_output ←
              mrlc_output.replace(gadget);
            // check if Toffoli gadget or AND gadget here
        end
        buffer_latency ← buffer_latency − 1;
    end
end
return gadget_tree, mrlc_levels;
```

Algorithm 4: `MRLCWrapper`

Input: $gadget_tree$, $mrlc_levels$, $target_latency$, d
Output: $gadget_tree$ with optimized gadgets and minimal latency across levels

```
latency_leastpossible ← number of levels in gadget_tree;
buffer_latency ← target_latency − latency_leastpossible;
if buffer_latency < 0 then
    print("Target latency cannot be met"); return;
end
Step1: Initial assignment of gadgets
for each gadget in gadget_tree do
    // Replace all gadgets with the least latency gadget in order d
    gadget ← gadget_least_latency;
    // check if Toffoli gadget or AND gadget here
end
Step2: Reduction of randomness using buffer latency
if buffer_latency > 0 then
    /*Arrange levels based on number of nodes (gadget placeholders such as Toffoli or AND)*/
    mrlc_levels ← construct_max_heap(gadget_tree);
    gadget_tree, mrlc_levels ←
      ReduceRandomnessWithBufferLatency(gadget_tree, mrlc_levels, buffer_latency, d);
end
return gadget_tree
```

6.1.1 Step1: Initial assignment of gadgets

Initially, the MRLC algorithm replaces all nodes with the lowest latency gadget by default. This provides the opportunity to maximize the $buffer_latency$ to iteratively minimize the randomness in the subsequent steps of the algorithm. If multiple gadgets have the same lowest latency, the gadget with the least $total_randomness$ and $total_latency$ is selected in that order. So if a gadget g with the least possible latency exists in $Gadgets_List$ such that $g.latency = 1$ ($g.latency = k$), then all gadget-nodes are assigned g, and the design has a latency of L ($k \times L$). This assignment will satisfy the target if $target_latency \leq L$ ($target_latency \leq k \times L$ respectively).

6.1.2 Step2: Reduction of randomness using $buffer_latency$

Step 1 assigns gadgets with the user's $target_latency$ already met. However, MRLC still needs to ensure that the design has the lowest possible randomness under that $target_latency$. The steps to do so are as follows:

- **Initial Buffer Latency:** In order to reduce the $total_randomness$, the $buffer_latency = target_latency - (k \times L)$ is calculated.

- **Sorting the levels:** The objective is to utilize the $buffer_latency$ to cause maximum reduction in the $total_randomness$. Increasing latency is most effective when applied uniformly across all nodes in a level. This is because the more nodes replaced by higher latency gadgets (with lower randomness due to the tradeoff), the lower the overall latency of the level. Hence, all the nodes in a level are replaced by the same higher latency gadgets using the $buffer_latency$. Consequently, increasing the latency of a level with a larger number of nodes causes a greater reduction in randomness. Therefore, MRLC sorts the levels in descending order based on the number of nodes in the levels. The algorithm picks the level with maximum nodes each time while updating the $buffer_latency$ and reducing the $randomness$ iteratively until no further utilization is possible. Prioritizing levels with the maximum number of nodes ensures that each unit of additional latency yields the greatest reduction in randomness, ensuring that *local decisions lead to globally optimal assignments, which is also facilitated by the level-wise structure of the gadget tree.*

Example 1: MRLC on $gadget_tree$ in Figure 4. Assume that the $Gadget_Library = \{HPC1, HPC2, HPC3, HPC2o, HPC3o\}$. Assume the $target_latency$ is 6, and the masking order is $d - 1 = 1$. For simplicity, let us assume there are no Toffoli gadget replacement candidates in the $gadget_tree$. We will replace all gadgets with HPC3 initially. This results in a design with latency = 3, randomness = 30, and area = 45 GC. Now our $buffer_latency = 3$. We construct the max heap and store it in $mrlc_levels = \{level1, level2, level3\}$. We replace all nodes in $level1$ with the lower $randomness$ gadget in our $Gadget_Library$ which are $\{HPC1, HPC2\}$. Among them, we choose $HPC2$ as it has a lower area (in GC). Similarly, we replace all gadgets in $level2$ and then $level3$ with HPC2. Resulting in a $total_randomness = 15$ with a $latency = 6$ and $buffer_latency = 0$, which is our optimal design. Our MRLC Algorithm presented in this section has been detailed in Algorithm 4.

7 Minimize Latency under Randomness Constraint (MLRC)

The MLRC problem states that, given a list of gadgets in $Gadgets_list$ such that for each gadget $g \in Gadgets_list$ with latency $g.latency$ and randomness $g.rand$ and a given $target_randomness$ by the user, we need to find the design(s) with the minimum latency satisfying the $target_randomness$.

7.1 MLRC Algorithm $d \geq 2$

MLRC involves the following three steps: Initial assignment of gadgets, Redistributing $buffer_randomness$ for low-latency gadget assignment, and Area optimization. In the subsections below, we detail the steps of the MLRC algorithm presented in Algorithm 6.

7.1.1 Step1: Initial assignment of gadgets

The output of MLRC is a gadget assignment that yields a masked design with minimum latency under the user-specified $target_randomness$. Thus, the algorithm must begin with an initial assignment that at least satisfies the $target_randomness$, thereby accelerating convergence to the optimal latency. This choice depends on $target_randomness$ and the number of nodes in the Gadget-tree $|V|$. For a design with $|V| \geq 6$, the following cases arise:

- **Case 1:** If the $target_randomness$ is lesser than the minimum randomness required ($|(d-1).V|$ which is the lowest possible existing randomness gadget for the $d^{th} order$), an optimized design does not exist.
- **Case 2:** If $|(d-1).V| \leq target_randomness < d.|V|$ each node can be assigned a gadget of randomness $(d-1)$ at-least. If such a gadget exists in the $Gadgets_list$, all nodes can be assigned with that gadget (e.g., HPC2/HPC2o for

$d = 2$). In this case, wherever the gadget is a Toffoli gadget candidate (as identified in Step 1 in Section 5, a Toffoli gadget is assigned (e.g., instead of HPC2, HPC2o is assigned). This uses exactly $|(d-1).V|$ randomness but gives high latency ($2L$) (Given the $Gadget_List = \{HPC1, HPC2, HPC3, HPC2o, HPC3o\}$ for any d the lowest randomness gadget has a randomness $d-1$) resulting in no $buffer_randomness = target_randomness - Gadget_tree.rand$.

- **Case 3:** Since $target_randomness \geq d.|V|$, a gadget with the lowest possible latency satisfying $gadget.rand \leq d$ can be chosen to replace all gadgets. Thus, in our case, for $d = 2$, HPC3 with $HPC3.latency = 1$ and $HPC3.rand = 2$ can be assigned to all gadgets. In this case, too, a Toffoli gadget candidate will be assigned a Toffoli gadget. No further optimization is needed.

These cases have been accounted for in lines $1-3$ and 8 in Algorithm 6.

Algorithm 5: `ReduceLatencywithBufferRandomness`

Input: *gadget_tree*, *mlrc_levels*, $buffer_randomness$, d
Output: Optimized *gadget_tree*

```
1  for each level in mlrc_levels do
2      if #gadgets in level ≥ buffer_randomness and latency(level) > 1 then
3          for each gadget in level do
4              if gadget.flipped ≠ 1 then
5                  /* replace gadget with the next low-latency gadget with least area*/
                    gadget_tree ← gadget_tree.replace(gadget); //check if Toffoli gadget or AND
                    gadget here
6              end
7              buffer_randomness ← buffer_randomness - 1;
8          end
9      end
10 end
11 return gadget_tree;
```

Algorithm 6: `MLRC: wrapper`

Input: $gadget_tree, target_randomness, Gadgets_List, d$
Output: MLRC execution with area optimization pass

```
1  Step1: Initial Assignment of Gadgets
2  mark all AND nodes with the least randomness gadget in Gadgets_List;
3  randomness_leastpossible ← (d − 1).|V| ; /* Assuming that there is at-least one gadget g in the
     Gadgets_List with g.rand = (d − 1) */
4  buffer_randomness ← target_randomness − randomness_leastpossible;
5  Step 2: Reduction of latency using Buffer Randomness if buffer_randomness > 0 then
6      mlrc_levels ← construct_min_heap(gadget_tree);
7      gadget_tree, mlrc_levels ← ReduceLatencywithBufferRandomness(gadget_tree, mlrc_levels,
         buffer_randomness, d);
8      return gadget_tree;
9  end
10 print("Target randomness cannot be met");
```

7.1.2 Step2: Re-distributing $buffer_randomness$ for low latency gadget assignment

For Case 2 in the initial assignment of gadgets in Section 7.1.1, they already satisfy the randomness budget of $target_randomness$. However, MLRC still has to ensure the lowest possible latency within that budget. To do so, the $buffer_randomness = target_randomness - (d-1).|V|$ needs to be effectively used to re-assign some of the gadgets with a higher randomness (lower latency) gadget, given the trade-off, to result in a lower latency design within the $target_randomness$. The following must be considered:

- **Level-wise latency reduction:** Given that a lower latency gadget would imply a higher randomness due to the trade-off (refer to Table in Figure 2), it is

most beneficial if all the gadgets of the level have the same latency. Otherwise, the increase in randomness due to decreased latency of some gadgets in a level will consume $buffer_randomness$ without reducing latency. This will reserve $buffer_randomness$ to reduce the *latency* of another level. Thus, it is crucial to determine which level to prioritize for latency reduction first. Once that is established, to result in a reduction in latency, the latency of all the gadgets in the level needs to be reduced to the same value of latency.

- **Level of interest:** Given a $buffer_randomness$, the best use of those extra random bits to reduce latency could be achieved if they can be used to cause the latency reduction in a maximum number of levels. This means it is better to choose many levels with fewer nodes each than fewer levels with more nodes each. *Thus, MLRC starts with the level with the least number of gadget nodes first, by extracting levels from a* `min_heap`. *The usage of a heap is essential here to prioritize levels that yield maximum randomness reduction per unit latency.* At any point, we call the level being extracted as the $current_level_of_interest$. The algorithm continues to reduce the latency of $current_level_of_interest$ by 1 in this manner, till $buffer_randomness = 0$ or it is impossible to reduce the latency of the $current_level_of_interest$ further as $|current_level_of_interest.nodes| > buffer_randomness$. This reduces the latency for a maximum number of levels with minimum additional randomness.

7.1.3 Step3: Area optimization

After the previous steps of the algorithm as detailed in Section 7.1.1 and Section 7.1.2, we would have obtained a design with minimum latency under randomness constraint. However, the area of such a design might not be optimal. In such a case, the following scenario must be handled:

- **Reduction of area within a level** After the MLRC process, if it's not feasible to replace a higher latency gadget (e.g., HPC2) with a lower latency gadget (e.g., HPC3) for all nodes in a level, MLRC will pick a subset of nodes in that level based on the $buffer_randomness$ and replace those with HPC3. This will utilize all remaining $buffer_randomness$ while reducing the area if the lower latency gadget has a lower area(e.g., $HPC3.area < HPC2.area$). This can be used when there is a gadget with the same or lower *latency* and lesser *area* available at the cost of extra *randomness*. Since the latency of a level is the maximum of the latencies of all the gadgets in the level, the level's latency will remain the same. The difference in latency in a level which we had avoided in Step 2 to conserve $buffer_randomness$ has now been used to reduce the area. This will result in a lesser area with better utilization of the $buffer_randomness$. Similar analysis applies to the case of Toffoli gadgets.

8 Complexity of Greedy MRLC and MLRC

$O(n)$ time is required to construct the $gadget_tree$ where n is the number of nodes (AND gadget or Toffoli gadget replacement). The MRLC algorithm in Algorithm 4 constructs a max-heap of pairs $(\#nodes_in_level, level_index)$, where the number of entries is at most $O(n)$. Thus, the heap construction requires $\mathcal{O}(n \log n)$ time. Then, the algorithm runs a loop for $buffer_latency$ times and the value of $0 \leq buffer_latency \leq L$ (L is the number of levels in the Gadget tree). The number of levels in the Gadget tree would be at most $O(n)$, thus the loop runs $O(n)$ times. Therefore, MRLC takes $T(n) = O(n) + O(n \log n) = O(n \log n)$ time.

The MLRC algorithm in Algorithm 6 constructs a min-heap of pairs *(#nodes in level, level index)*, where the heap construction requires $\mathcal{O}(n \log n)$ time. Then, the algorithm runs a loop for $buffer_randomness$ times and the value of $0 \leq buffer_randomness \leq n$

nodes in the Gadget tree. Inside the loop, each gadget is replaced with its behavioral description in $O(1)$ time. Therefore, $T(n) = O(n) + O(n \log n) = O(n \log n)$.

9 Optimizations with Asymmetric Gadget Compositions

While MRLC and MLRC optimize at the gadget selection level, they do not account for structural inefficiencies introduced by asymmetric gadgets. Additional optimizations can exploit gadget structure to further reduce the randomness and latency of gadget-based masked hardware. Specifically for gadgets (e.g., HPC1, HPC2, HPC2o) with unequal latency along each path to the output. Consider the HPC2 gadget in Figure 1a: for each output, one path uses two registers and the other uses one, resulting in *asymmetric latency*. Notice that this *asymmetry* is a consequence of which of the configurations from Figure 6 the gadget is in. Additional pipeline registers are required to result in a correct register-balanced design [STK24] in such a scenario with un-optimal *packing*. To improve such *packing*, the gadgets themselves can be replaced (with a symmetric gadget) or their orientation altered. Under favorable conditions, such a transformation may reduce latency and randomness.

Motivating Example: Consider the example *Gadget_tree* in Figure 5 (a). It performs the following operations:

```
Inputs:  a, b, c
Outputs:  p, q
p = b & c
q = p & a
```

Consider, $Gadgets_List = \{HPC2, HPC3\}$. We have performed MRLC on the *Gadget_tree* with a $target_latency = 2$. For a latency budget of 2, MRLC will choose HPC3 for p and q, giving the design in Figure 5(b) for an order of masking $d = 2$, with a randomness cost of 4. However, due to the asymmetry, one can instead instantiate p with HPC3 and q with HPC2, where one chooses p as the gadget input with output latency 1 and q as the input with output latency 2. This solution requires 3 random bits as shown in Figure 5(c), which means there exists a solution with lower randomness than the MRLC solution.

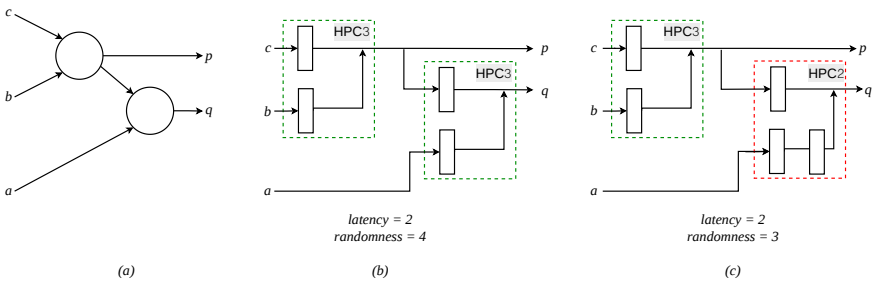


Figure 5: Motivating example for asymmetric gadgets.

In order to deal with such randomness and latency reductions solely due to asymmetry of latency in gadgets (with no additional latency or randomness cost), we incorporate the following transformations into our MLRC and MRLC flow:

- *Latency Optimization:* **Transform 1** Reversing the inputs of two such asymmetric gadgets results in a reduction in latency (Figure 7(a)(b)(c)).
- *Randomness Optimization:* Composition of two gadgets can result in a randomness optimization opportunity arising under the following conditions:
  - **Transform 2** When an asymmetric and a symmetric gadget are composed in series with the symmetric gadget dependent on the asymmetric one, replacing

the dependent symmetric gadget with an asymmetric gadget with the inputs flipped results in a reduction of randomness (Figure 7(d)(e)(f)).

- **Transform 3** When a symmetric and an asymmetric gadget are composed in series, with the asymmetric gadget dependent on the symmetric one, flipping the asymmetric gadget and replacing the symmetric gadget with an asymmetric one leads to a reduction in randomness. (Figure 7 (j)(k)(l)).
- **Transform 4** When two symmetric gadgets are composed in series, replacing the dependent symmetric gadget with an asymmetric gadget with the inputs flipped results in a reduction of randomness along with a reduction of balancing register(s) (Figure 7(g)(h)(i)).

Asymmetric gadgets introduce path imbalance, which manifests as additional balancing registers. By strategically *packing* asymmetries across adjacent gadgets, these registers can be eliminated, effectively reducing both area and latency. This also comes with additional randomness reduction without any latency cost for a trade-off. Note that in all the above cases, the dependent gadget should have at least one pipeline register available to accommodate the asymmetric gadget. For example, in Figure 7(a), the path from c to d should have a minimum of one pipeline register to accommodate the extra register on the path introduced by the flipping, as in Figure 7(c). Also, in all four cases, the optimized design (after the transform) has fewer pipeline registers (i.e., registers not required by masking).

9.1 Transform 1

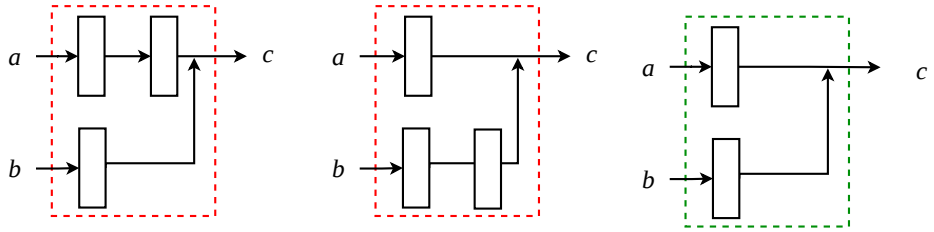
In Algorithms 7 and 8, we present the algorithms to perform the aforementioned transforms. Consider Algorithm 7 that performs Transform 1 to flip the second gadget in an $asymmetric-asymmetric$ pattern. This asymmetric pattern can be detected via a simple heuristic called *asymmetry* of a gadget, which calculates the difference in the number of registers from the primary inputs to output for both output paths of a gadget in the gadget tree, the registers being contributed by the gadget itself and the gadget prior to it in the path π. For example, in the Figure 7(a) without the purple registers, the asymmetry of the second gadget is 3, and the asymmetry of the first gadget is 1. It follows that a gadget with a higher value of *asymmetry* would have more number of asymmetric gadgets in an unoptimal orientation (dependent gadget in Figure 7(a)) in its path from the input. Hence, it would be most beneficial to start at that gadget and continue down the path to the primary inputs, alternatively flipping gadgets if they are asymmetric too. This continues until all the gadgets in the design have an asymmetry ≤ 1, that is, the gadget itself is asymmetric ($asymmetry = 1$) or symmetric, and the design is maximally packed.

9.2 Transforms 2, 3, and 4

Transforms 2, 3, and 4 involve symmetric gadgets with asymmetric gadgets or with another symmetric gadget (Transform 3). In both Transform 2 and 3, the second gadget is a symmetric gadget, and the transformation just inserts a flipped asymmetric gadget in its place (lines 6 - 7 in Algorithm 8). Similarly, for Transform 3, the pattern is scanned from the input, and once detected, the asymmetric gadget is flipped, and the symmetric gadget is replaced by an asymmetric gadget.

Assumptions In using asymmetric gadgets in our algorithms, we assume the following:

- All the asymmetric gadgets in the gadget library are initially placed in their unrotated (un-flipped) form. We call this the *default configuration* of an asymmetric gadget. In this gadget library, we consider asymmetric gadgets of latency 2 (HPC1, HPC2, HPC2o), their default configuration is shown in Figure 6a, and the flipped configuration is shown in Figure 6b. For the symmetric gadgets considered, the default configuration, as shown in Figure 6c, is the only configuration.



(a) asymmetric default *(b) asymmetric flipped* *(c) symmetric default*

Figure 6: Default and flipped configuration of symmetric and asymmetric gadgets.

- A gadget is *not* in its default configuration as a result of Transforms 1 - 4 being applied.
- Once a transformation pass has been applied to it, the gadget is considered as *flipped* and subsequent transforms must not alter its orientation back to *default*.

Algorithm 7: `Transform 1`

Input : *gadget_tree*
Output : Optimized *gadget_tree*

Step 1: Compute asymmetry for each gate for *each gadget* $g \in$ *gadget_tree in topological order* **do**
  depth($g.\text{in}_1$) ← number of registers from $g.\text{in}_1$ to primary input
  depth($g.\text{in}_2$) ← number of registers from $g.\text{in}_2$ to primary input
  g.asymmetry ← |depth($g.\text{in}_1$) − depth($g.\text{in}_2$)|
end
Step 2: Iteratively process gadgets by decreasing asymmetry
while $\exists g : g.asymmetry > 1$ *and gadget not processed* **do**
  g^* ← arg max$_g$ g.asymmetry; //gadget with max asymmetry in the design
  Step 3a: Flip g^*
  Flip g^*
  Mark g^*.flipped ← 1
  Step 3b. Follow paths from inputs of g^* **to primary inputs**
  for *each path* π *from inputs of* g^* *toward primary inputs* **do**
    level$_0$ ← topological level of g^*
    for *each gate* h *on* π *(in reverse topological order)* **do**
      ℓ_h ← topological level of h; $d \leftarrow |\ell_h - \text{level}_0|$
      if $h.asymmetry > 1$ ***and*** $d \equiv 0 \pmod 2$ *and* $h.flipped = 0$ **then**
        Flip h ; h.flipped ← 1
      end
      else if $h.asymmetry = 0$ **then**
        `// Skip non-asymmetric gates; do not flip`
        Continue to the next gate on the path
      end
    end
  end
  `// Step 4b: Mark` g^* `as processed`
  g^*.processed ← 1
end
return *gadget_tree with flipped indicators*

9.3 MRLC: Overall Algorithm with Asymmetric Gadgets

In this section, we present the top-level MRLC DSE algorithm, including the transforms in Section 9.1 and Section 9.2. Algorithm 9 takes as input the unmasked design, where Toffoli gadget placement is optimized. After the linear gates to be included with Toffoli gadgets have been identified, the rest of the linear gates are assigned their corresponding linear gadgets (Algorithm 1, Algorithm 2). Thereafter, the Toffoli gadget placement candidates and the AND gadget replacement candidates are used to construct a *dependency* tree

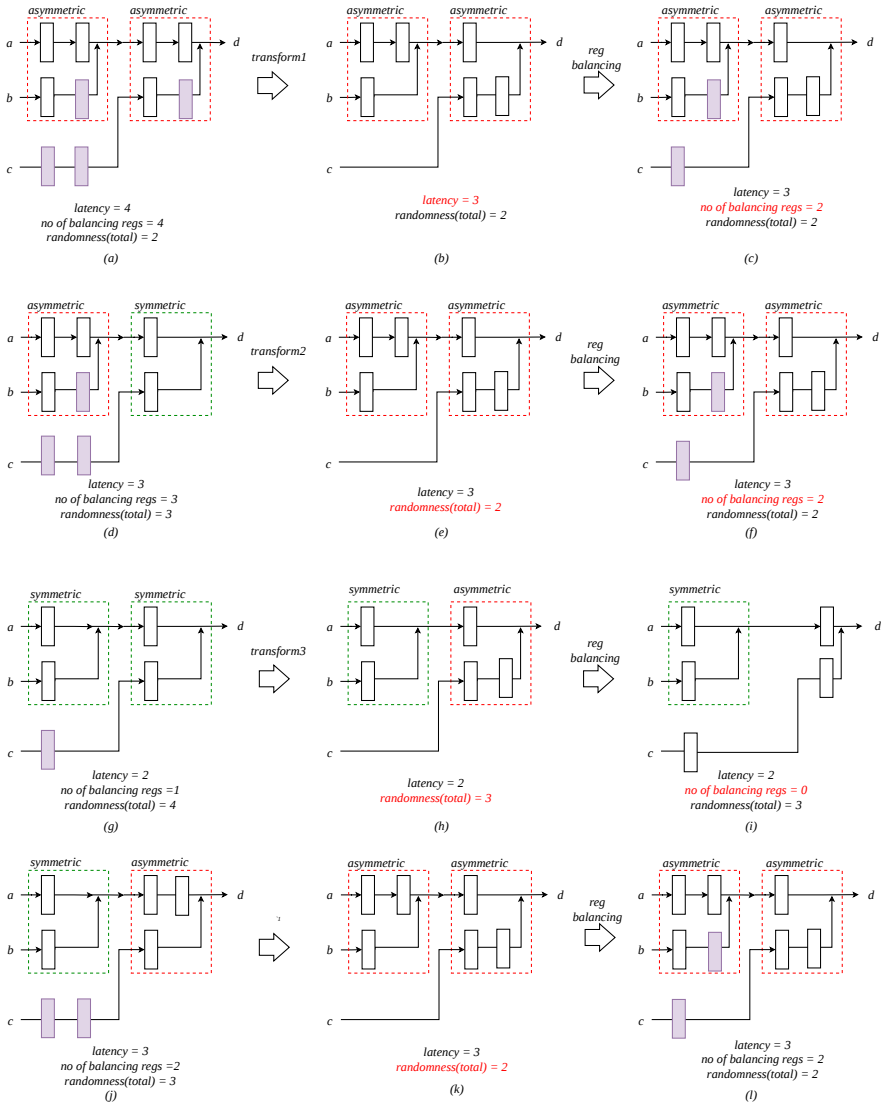


Figure 7: Asymmetric Gadgets in Composition: Transform1 (a) (b) (c), Transform2 (d)(e)(f), Transform3 (g)(h)(i), Transform4 (j)(k)(l).

Algorithm 8: `Transform 2,3,4`

```
   Input  : gadget_tree
   Output : Optimized gadget_tree
 1 for each level ∈ Gadget_tree in topological order starting from input (excluding input level) do
 2     for each gadget in level do
 3         gadget.asymmetry ← |gadget.in1.depth − gadget.in2.depth|;
 4         if gadget is symmetric and gadget.asymmetry ≥ 1 then
 5             /* replace gadget with appropriate gadget*/
 6             mlrc_output ← mlrc_output.replace(gadget);
 7             gadget.flipped ← 1;
 8         end
 9     end
10 end
```

called the *gadget_tree*. Initially, the worst-case randomness design with the least latency is sought. If such a design is found, the leftover latency ($buffer_latency$) (Algorithm 3) and the transformations (Algorithm 8) are applied to reduce randomness further. Finally, Transform 1 (Algorithm 7) is applied on this modified *gadget_tree* with the remaining unoptimized levels from the randomness reduction using buffer latency (Algorithm 3). This transformation leads to a reduction in latency, which in turn facilitates a reduction in randomness by utilizing the buffer latency (as in line 11). This is performed in a loop, reducing randomness until no further optimization is possible. The optimized gadget tree is hence used to obtain the masked RTL, ensuring minimum latency and registers via MaskedHLS [STK24].

9.4 MLRC Overall Algorithm with Asymmetric Gadgets

The MLRC Algorithm in Algorithm 10 is a dual of the MRLC Algorithm and has the same pre-processing steps from Steps 1 to 3. Thereafter, an initial pass of the MLRC algorithm (Algorithm 6) is directly applied to obtain a design with minimum latency under the randomness constraint. Note that with only the worst case (least randomness gadgets to all) assignment, the application of transformations 2 - 4 is not possible, as they need at least one symmetric gadget. Also, since an asymmetric gadget will always be the gadget with more latency than 1, the initial MLRC pass (Algorithm 6) could use the $buffer_randomness$ from the MLRC pass to reduce latency first, leaving the structural latency reductions till later for Transform 1. Hence, only after MLRC using Algorithm 6 has been applied (Algorithm 10 line 4), we apply the transform using Algorithm 10 line 4 onwards.

10 Including COMAR in the DSE

COMAR [KM23] is a 1st-order masking gadget with unique properties that require special handling via the DSE approaches presented in this paper.

10.1 MRLC with COMAR

For MRLC with COMAR in the *Gadgets_List* the Algorithm 9 fails to account for COMAR's *unusual* randomness-reuse property that allows it to be instantiated any number of times in the design using the same 6 refresh bits. In the next Sub-Sections, we discuss this in detail, along with a solution.

10.1.1 Why Greedy MRLC Fails with COMAR

Figure 1c represents the COMAR gadget for $c = a \otimes b$. COMAR uses six random bits in total, which is larger than the HPC1, HPC2, and HPC3 2-input AND gadgets for $d = 2$. However, *all instantiated 2-input COMAR-AND gadgets in a circuit can use the same six random masks*. This contradicts the assumptions underlying the greedy MRLC (Algorithm 4) optimization strategy, which assumes that each gadget's randomness cost is independent, enabling locally optimal gadget assignments at each level. Our MRLC Algorithm in

Algorithm 9: `MRLC`

Input: *unmasked_design*, *Gadgets_List*, *target_latency*, *d*
Output: Optimized *masked_design*

Step 1: Identify Toffoli gadget candidates and mark them in the AST;
Step 2: Perform linear gadget assignment;
Step 3: Construct *gadget_tree*;
Step 4: Initialize gadgets in *gadget_tree*;
gadget_tree ← `ReplaceAllWithLeastLatency`(*gadget_tree*); `// Replace all gadgets with minimum latency versions`
Step 5: Compute buffer latency;
buffer_latency ← *target_latency* − *gadget_tree.latency*;
Step 6: Reduce randomness using Transforms 2, 3, 4;
gadget_tree ← `Transforms_234`(*gadget_tree*);
Step 7: Reduce randomness using buffer latency;
(*gadget_tree*, *mrlc_levels*) ← `ReduceRandomnesswithBufferLatency`(*gadget_tree*, *buffer_latency*);
Step 8: Transform 1 (latency reduction pass);
repeat
  (*gadget_tree*, *mrlc_levels*) ← `Transform1`(*gadget_tree*, *mrlc_levels*); `// See Algorithm 7`
  buffer_latency ← *target_latency* − *gadget_tree.latency*; `// Recompute buffer latency`
  gadget_tree ← `ReduceRandomnesswithBufferLatency`(*gadget_tree*, *mrlc_levels*, *buffer_latency*);
until *(buffer_latency* = 0*)* ***or*** *(no more optimization possible)*;
Step 9: Register balancing and synthesis;
masked_design ← `MaskedHLS [STK24]`(*gadget_tree*);
return *masked_design*

Algorithm 4 uses buffer latency to reduce randomness across gadgets, level-wise. However, with COMAR, selecting a higher-latency gadget (COMAR) early in the design may incur no additional randomness cost for subsequent instances if only the COMAR gadget is used to replace all AND nodes throughout the circuit. Consequently, a locally unoptimal choice (e.g., higher latency, higher per-instance randomness) can lead to a globally optimal solution through randomness reuse. To address that, adding a $COMAR_used$ flag would not work either, as at each level, compared to the randomness cost of all other gadgets at the 1st-order, $COMAR$ will always have a higher randomness cost. Hence, the algorithm will never pick $COMAR$. Thus, a greedy strategy is unable to perform MRLC with the COMAR gadget in the *Gadgets_List*. Therefore, we need a new algorithm to perform MRLC using COMAR in the *Gadgets_List*.

However, the problem retains optimal substructure: the optimal gadget assignment for levels 1 to i, for each possible pair of latency consumed j and COMAR usage flag $k \in \{0, 1\}$ being indicative of whether $COMAR$ has been a part of any previous solution corresponding to levels $x < i$, can be extended to level $i + 1$ by simply choosing the best gadget (least randomness costing) at level $i + 1$ based on the buffer latency and the current value of k. This admits a dynamic programming (DP) formulation that tracks i, j, and k, exploring all gadget choices per level while paying COMAR's shared randomness cost at most once. Below, we present the DP-based optimization for the case where COMAR is in *Gadgets_List*.

Algorithm 10: `MLRC`

Input: *unmasked_design*, *Gadgets_List*, *target_randomness*, d
Output: Optimized *mlrc_output*

Step 1: Identify Toffoli gadget candidates and mark them in the AST;
Step 2: Perform linear gadget assignment;
Step 3: Construct *gadget_tree*;
Step 4: Initialization and one MLRC pass;
mlrc_output ← `MLRC_Wrapper`(*target_randomness*); `// See Algorithm 6`
previous_randomness ← *mlrc_output.randomness*;
if *mlrc_output.randomness* ≤ *target_randomness* **then**
  Step 5: Apply Transforms 2, 3, 4;
  mlrc_output ← `Transforms_234`(*mlrc_output*);
  Step 6: Compute buffer randomness;
  buffer_randomness ← *previous_randomness* − *mlrc_output.randomness*;
  Step 7: Iterative optimization;
  while *(buffer_randomness* > 0*)* ***and*** *(not converged)* **do**
    mlrc_output ← `ReduceLatencyWithBufferRandomness`(*mlrc_output*, *buffer_randomness*)`/*See Algorithm 5.*/`;
    buffer_randomness ← *previous_randomness* − *mlrc_output.randomness*;
  end
  Step 8: Transform 1 (latency reduction pass);
  mlrc_output ← `Transform1`(*mlrc_output*); `// See Algorithm 7`
  return *mlrc_output*;
end
Step 9: Register balancing and synthesis;
masked_design ← `MaskedHLS [STK24]`(*gadget_tree*);

10.1.2 MRLC: Dynamic Programming Approach ($d = 2$)

MRLC can be formulated as a dynamic programming problem with L levels and a latency budget of $target_latency$. Let $\mathrm{dp}[i][j][k]$ denote the minimum randomness gadget assignment for the first i levels (from input) with a latency $j \leq target_latency$, $i \in [0, L], j \in [0, target_latency]$ and $k \in \{0, 1\}$ indicating whether $COMAR$ has been assigned before.

For the solution at level i, if we select a gadget $g \in Gadgets_list$ with latency cost $g.latency$, we must have spent $j - g.latency\ buffer_latency$ up to level $i-1$ (all previous levels). Thus, the total randomness becomes: (randomness from levels 1 to $i-1$) + ($g.rand$). For MRLC, our objective is to try to minimize this quantity.

For a non-COMAR gadget g:

$$\mathrm{dp}[i][j][k] = \min_{g \notin \{COMAR\}} \{\mathrm{dp}[i-1][j - g.latency][k] + g.rand\}$$

For COMAR:

$$\mathrm{dp}[i][j][1] = \min \begin{cases} \mathrm{dp}[i-1][j-2][0] + 6 & \text{(first use of COMAR)} \\ \mathrm{dp}[i-1][j-2][1] + 0 & \text{(already instantiated COMAR)} \end{cases}$$

We use this to fill the table for our dynamic programming problem for each level i to finally obtain for our design the dp[i][j][k] for level $i = L$.

Extracting gadget assignment: MRLC solution To extract the actual gadget assignment, a data structure is maintained that tracks the selection of gadgets that led to the minimum value. This allows the algorithm to backtrack from dp$[L][j^*][k^*]$ where j^* and k^* correspond to the optimal solution for the DP at the last level L. Thus our MRLC Algorithm 4 will be replaced by this DP formulation in case COMAR is used in the *Gadgets_List*.

Complexity Analysis The complexity of this DP MRLC algorithm is given as follows: $\mathcal{O}(L \cdot target_latency \cdot |\mathcal{G}|)$, where L is the number of levels, $target_latency$ is the target latency, and $|\mathcal{G}|$ is the size of the gadget library.

10.2 MLRC with COMAR

In contrast to MRLC, including COMAR in the *Gadgets_List* does not require special handling or alternate algorithms for our Greedy MLRC as in Algorithm 10. The following Sub-Sections discuss this in detail.

10.2.1 Why Greedy MLRC works with COMAR

In MLRC, trading $buffer_randomness$ with latency, we never face a situation where it is rewarding to deviate from the greedy strategy of using $buffer_randomness$ optimally to reduce latency at a level, even with COMAR in the *gadgets_list*. This is because COMAR has a fixed randomness across all instances in the design but incurs a latency of 2 per instance.

10.2.2 Initial gadget assignment

In MLRC, the choice of the initial gadget assignment depends on $target_randomness$, the number of nodes in the $gadget_tree = |V|$, and, owing to the re-usability of randomness in COMAR, whether $COMAR \in Gadgets_List$. In case $COMAR \in Gadgets_List$, then we must consider the following case: If $target_randomness \geq 6$ but $< |V|$, all nodes can be replaced with COMAR. This results in the least possible randomness. Any other gadget assignment will have at least $total_randomness = |V|$ for order 1.

10.2.3 Area overhead of COMAR

In the baseline MLRC algorithm (Algorithm 6), we prioritize using less randomness; hence, it should logically prioritize COMAR over any other gadget with a latency of 2. Even though COMAR offers a significant advantage in terms of randomness, its area overhead is quite high (almost 3× that of HPC2, say). Thus, for a design that prioritizes optimal area, COMAR might not be the right choice. For cases where the $buffer_randomness > 6$, the algorithm must use this $buffer_randomness$ to replace most of the COMAR gadgets (in the already optimal assignment) with gadgets having less area than COMAR. This will increase the $total_randomness$ of the design while still staying within the $target_randomness$, since we are using the $buffer_randomness$ left after MLRC. This could be used for any two gadgets with the same *latency* but different *area*.

11 Register Balancing Optimization using MaskedHLS

For a masked pipelined circuit to function correctly, in addition to the mandatory masking registers, extra registers must be inserted into parallel paths to ensure correct pipelining. Consider the design in Figure 8(a) with mandatory registers that must be inserted at the locations shown by the red rectangles. To ensure the design is correctly pipelined, the number of registers at each input path to a gate must be equal: a property referred to as *balancing*. To ensure that, *balancing registers* shown as yellow rectangles in Figure 8(b) need to be inserted. A naive way of *balancing* the design in Figure 8(a) would be as shown in Figure 8(b) with a total of 8 regs and a latency of 2. However, as shown in Figure 8(c), with a total of 4 regs and a latency of 1, a correctly *balanced* pipelined design

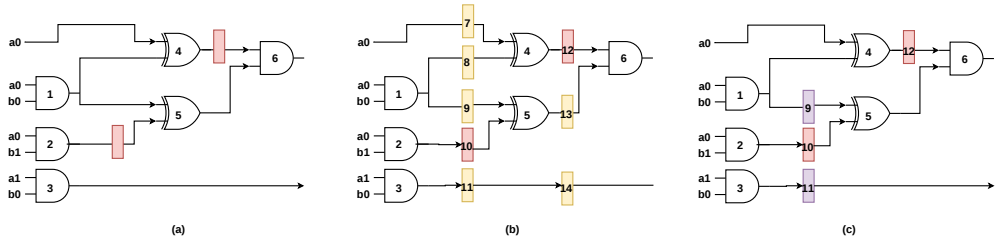


Figure 8: Example to demonstrate the need for optimal register balancing.

can be obtained that contains the mandatory registers (in red) and balancing-registers (in purple). The process of automatically and optimally placing these *balancing registers* is called register balancing [STK24]. Register-balancing imposes a strict *synchronization constraint* on the design space, wherein a minimum number of extra registers, apart from the mandatory ones (due to glitch-robust masking, for instance), is required to be inserted in parallel paths of the design.

At Step 4 in Section 5, TETRIS's *Gadget_Library* includes gadgets with mandatory registers in their specifications. Enforcing these balancing registers at the gadget level is therefore not required by masking and could lead to latency overhead as demonstrated in Figure 8. Hence, at Step 4, after the gadget assignment, TETRIS replaces gadgets in the *Gadget_tree* with only the mandatory registers (for masking), and adds the pipeline registers after the DSE via *register-balancing*on the overall design using MaskedHLS [STK24] in Step 5. This automatically ensures there is minimal latency or register overhead due to *pipelined* registers instead of relying on local pattern-matching with optimal register placement for reducing register usage in an overall masked design. The register-balancing step serves as the final motivation of the paper, as in Section 4.

Lemma 2. *Let D^* be an optimal design produced by the MRLC algorithm for a given latency budget L. If $\mathcal{R}(D)$ represents the randomness cost and $\mathcal{B}(D)$ is the number of balancing registers, then D^* satisfies:*

$$\mathcal{B}(D^*) = \min\{\mathcal{B}(D) \mid \mathit{Latency}(D) \leq L\}$$

Lemma 3. *Let D^* be an optimal design produced by the MLRC algorithm for a given randomness budget R. The design D^* achieves the global minimum latency L_{min} only if the number of balancing registers $\mathcal{B}(D^*)$ is the theoretical minimum required to synchronize the selected set of gadgets under the budget R.*

12 Combining MLRC and MRLC to meet both Randomness and Latency Requirements

TETRIS takes one target optimization parameter (either randomness or latency, not both) for the scenario where a user wants a design that meets the target while minimizing the other parameter. The MLRC and MRLC algorithms in TETRIS are therefore independent and cannot be combined for the given optimization objective considered in the paper.

However, in the scenario where both randomness and latency targets are provided (r_{max} and l_{max}) and a design meeting both simultaneously is desired, MLRC and (a portion of) MRLC can be combined to obtain a design satisfying both budgets when feasible. Consider Algorithm 11 that illustrates this combined flow as a use case. For simplicity, we do not include Transforms 1, 2, 3, and 4 in this workflow. It takes as input the *Gadget_tree*, D, the randomness budget r_{max}, the latency budget l_{max}, with the objective of finding a gadget-masked design *satisfying* both l_{max} and r_{max}. Initially, the MLRC algorithm in Algorithm 6 is called with the target randomness r_{max}. The result

of MLRC is a design with randomness $\leq r_{max}$ and a latency L'. Thus, if $L' \leq l_{max}$, then the design satisfying both r_{max} and l_{max} is feasible, and the Algorithm outputs feasible and $\langle R', L' \rangle$ as MLRC already finds the design with the least latency. Now, in case $L' > l_{max}$, and there is still a buffer randomness of $r_{max} - R'$, it could be used via the `Reduce_latency_with_randomness_buffer` (Algorithm 5) sub-function of MRLC to reduce the latency further than L'. If such a reduction is successful, the design has a latency $L'' \leq l_{max}$ and randomness $R'' \leq r_{max}$. Otherwise, the target latency and randomness can not be *simultaneously* met. In the general case, a similar Algorithm may be obtained using the `MRLCWrapper` in Algorithm 4 and the `Reduce_randomness_with_latency_buffer` sub-function from Algorithm 3.

Algorithm 11: DSE(D, r_{max}, l_{max},d) – Joint MLRC-MRLC Algorithm

Input: Gadget tree D, randomness budget r_{max}, latency budget l_{max}, masking order d
Output: $\langle r, l \rangle$: achieved randomness and latency

$D' \leftarrow$ `MLRCWrapper`$(D, r_{max}, Gadgets_List, d)$ `/*See Algorithm 6*/`;
$L' \leftarrow D'.latency$;
$R' \leftarrow D'.randomness$;
if $L' \leq l_{max}$ **then**
  return `feasible` $\langle r_{max}, L' \rangle$;
else
  if $R' < r_{max}$ **then**
    $\langle R'', L'' \rangle \leftarrow$ `REDUCE_LATENCY_WITH_RANDOMNESS_BUFFER`$(D, r_{max} - R')$`/*See Algorithm 5*/`;
    if $L'' \leq l_{max}$ **then**
      return `feasible` $\langle R'', L'' \rangle$;
    else
      return `infeasible`;
    end
  else
    return `infeasible`;
  end
end

13 Experimental Evaluation

This section presents the results of using our DSE framework, comprising TETRIS's MLRC and MRLC algorithms, on three S-boxes and one full (16 rounds) first-order masked AES-128 with Boyar and Peralta's S-box. We present the benchmarks and their details, followed by the implementation details. We show the efficiency of the DSE framework in generating optimal masked Verilog. We also compare, wherever possible, our DSE performance with COMPRESS [CGM+24], an automated MRLC-like optimizer for masked hardware.

A note on comparison with COMPRESS: In certain cases, COMPRESS uses pre-optimized sub-circuits (via randomness-sharing [BBP+16], for example) for their DSE (as in Canright's S-box with the $G16_mult$ modules) to reduce randomness, area costs significantly, and also solver time (due to the pre-optimization). Since TETRIS performs only gadget-level DSE, we cannot make a fair comparison with COMPRESS in this case. TETRIS focuses on exploiting the *structure* of gadget-based masked hardware to achieve better optimization via trade-offs, yielding a design that serves as the basis for such optimizations.

13.1 Benchmark details

Since masking the linear parts of a block cipher is trivial, we chose to evaluate our DSE framework on the non-linear parts of block ciphers, namely S-boxes. We evaluate our MLRC DSE algorithm and our MRLC DSE algorithm on three S-boxes: (a) AES Boyar

and Peralta's S-box [BP12], (b) AES-256 Canright's S-box [Can05], and (c) SKINNY S-box (8 bit) [BJK+16]. Details of the unmasked implementations of these benchmarks are presented in Table 1. It may be noted that the unmasked versions of these designs have 4, 4, and 2 levels of nodes, respectively, in the gadget-tree constructed for the DSE process.

Table 1: Comparison of different S-box designs.

S-box (reference)	**AND levels**	**Total AND gates**
AES - Boyar and Peralta's S-box [BP12]	4	34
AES - Canright's S-box [Can05]	4	36
SKINNY 8 bit S-box [BJK+16]	4	8

Table 2: MRLC comparison with COMPRESS: Boyar-Peralta's S-box Results

$d-1$	Latency	Design	#Random	Area (kGE) w/o rand	Area (kGE) w/ rand	Time (sec)	Speedup
1	4	Tetris	46	2.83	4.64	0.063	—
		Comp: Base		3.53	5.34	1.01	15.9×
		Comp: Sep		3.24	5.05	1.63	25.6×
		Comp: Opt		**2.78**	**4.59**	2.35	**36.9**×
2	4	Tetris	138	**6.68**	**12.11**	0.085	—
		Comp: Base		7.91	13.34	1.12	13.2×
		Comp: Sep		7.47	12.9	1.68	19.8×
		Comp: Opt		6.79	12.22	2.28	**26.8**×
3	4	Tetris	276	**12.16**	**23.03**	0.08	—
		Comp: Base		14.03	24.89	1.11	13.9×
		Comp: Sep		13.44	24.3	1.76	22.0×
		Comp: Opt		12.96	23.82	2.27	**28.4**×
4	4	Tetris	460	**19.77**	**37.89**	0.12	—
		Comp: Base		21.89	40	1.1	9.2×
		Comp: Sep		21.15	39.26	1.64	13.7×
		Comp: Opt		20.45	38.55	2.28	**19.0**×
1	5	Tetris	37	3.38	4.83	0.054	—
		Tetris: w/ comar	34	3.48	4.81	0.099	—
		Comp: Base	37	3.81	5.26	1.52	28.1×
		Comp: Sep	37	3.51	4.97	3.99	73.9×
		Comp: Opt	37	**3.08**	**4.54**	5.82	**107.8**×
2	5	Tetris	111	7.97	12.34	0.06	—
		Comp: Base		8.54	12.91	1.61	26.8×
		Comp: Sep		8.1	12.47	4.09	68.2×
		Comp: Opt		**7.47**	**11.84**	5.86	**97.7**×
3	5	Tetris	222	14.48	23.23	0.86	—
		Comp: Base		15.17	23.91	1.54	1.8×
		Comp: Sep		14.58	23.32	3.85	4.5×
		Comp: Opt		**14.14**	**22.87**	5.75	**6.7**×
4	5	Tetris	370	22.89	37.45	0.12	—
		Comp: Base		23.68	38.25	1.63	13.6×
		Comp: Sep		22.95	37.51	3.52	29.3×
		Comp: Opt		**22.24**	**36.8**	5.79	**48.2**×
1	6	Tetris	34	3.85	5.19	0.055	—
		Tetris: w/ comar	31	3.71	4.93	0.053	—
		Comp: Base	34	4.06	5.4	2.53	46.0×
		Comp: Sep	34	3.77	5.11	8.69	158.0×
		Comp: Opt	34	**3.34**	**4.67**	17.43	**316.9**×
2	6	Tetris	102	9.04	13.05	0.14	—
		Comp: Base		9.0	13.01	2.41	17.2×
		Comp: Sep		8.56	12.57	8.58	61.3×
		Comp: Opt		**7.92**	**11.94**	14.79	**105.6**×
3	6	Tetris	204	16.4	24.43	0.94	—
		Comp: Base		15.87	23.9	2.34	2.5×
		Comp: Sep		15.28	23.9	8.25	8.8×
		Comp: Opt		**14.88**	**22.91**	32.28	**34.3**×
4	6	Tetris	340	25.9	39.29	0.98	—
		Comp: Base		24.69	38.07	2.33	2.4×
		Comp: Sep		23.95	37.33	6.88	7.0×
		Comp: Opt		**23.37**	**36.65**	13.83	**14.1**×

13.2 Implementation Details

TETRIS has been implemented in Python. The input behavioral description in C is parsed using Pycparser [Ben12]. The final masked RTL (Verilog) output is generated via the **open source** MaskedHLS tool[1] using the Pyverilog [TY15] parser. Given an input unmasked C

[1] GitHub:https://github.com/nilotpolas/MaskedHLS_LP

Table 3: MRLC: Canright_sbox Results

$d-1$	Latency	Design	#Random	Area (kGE) w/o rand	Area (kGE) w/ rand	Time (sec)
1	4	Tetris:	48	2.81	4.70	0.32
		Tetris: rand_reuse	36	2.80	4.21	0.38
2	4	Tetris:	130	6.66	11.78	0.67
		Tetris: rand_reuse	108	6.65	10.90	0.74
3	4	Tetris:	276	12.13	23.00	0.83
		Tetris: rand_reuse	216	12.12	20.6	0.78
4	4	Tetris:	460	18.57	36.69	0.83
		Tetris: rand_reuse	360	18.54	32.72	0.88
1	5	Tetris:	37	3.35	4.80	0.36
		Tetris: w/ comar	36	3.57	4.98	0.37
		Tetris: rand_reuse	35	3.32	4.69	0.42
2	5	Tetris:	111	7.91	12.28	0.41
		Tetris: rand_reuse	105	7.90	12.03	0.48
3	5	Tetris:	222	14.39	23.13	0.54
		Tetris: rand_reuse	210	14.35	22.6	0.56
4	5	Tetris:	370	22.03	36.60	0.61
		Tetris: rand_reuse	350	22.00	35.79	0.72
1	6	Tetris:	36	3.75	5.16	0.37
		Tetris: w/ comar	31	3.82	5.04	0.37
2	6	Tetris:	108	8.75	13.00	0.45
3	6	Tetris:	216	15.81	24.32	0.56
4	6	Tetris:	360	24.13	38.31	0.66

code, the end-to-end DSE and HLS flow consisting of TETRIS and MaskedHLS produces an optimal masked RTL based on the user's *target_randomness* or *target_latency*. All benchmark results have been generated by running the DSE framework on a 6-core Intel i7-8700 CPU operating at 3.20GHz.

13.3 MRLC Results and comparison with COMPRESS [CGM+24]

The MRLC algorithm was used to generate randomness-minimal designs for AES Canright's, AES Boyar and Peralta's, and SKINNY Sbox-es given a *target_latency* and compare corresponding designs with the COMPRESS tool at their Base, Sep and Opt strategies [CGM+24]. In each of the comparison Tables 2, 3, 4, the security order is $d-1$, one less than the number of shares, and the latency is in cycles. The area of the designs generated via TETRIS is generated with the same synthesis library (NanGate45) using the same Yosys script as COMPRESS [Cas23]. The Area for TETRIS is sans the area required for generating the random numbers (e.g., via an unrolled trivium [CMM+24]). The time required for MRLC is considered as the Time (sec) for TETRIS which is compared with the solver time field in COMPRESS results [Cas23].

13.3.1 MRLC vs COMPRESS on Boyar Peralta's AES S-box:

Table 2 lists the result of comparing our MRLC algorithm's results with COMPRESS on Boyar Peralta's S-box for orders $(d-1)$ ranging from 1 to 4 for *target_latency* ranging from 4 (number of levels) to 6. For all the designs with the primary motive of reducing randomness, we are able to match COMPRESS exactly the number of random bits for masking corresponding to each *target_latency* across all masking orders. However, in some cases our tool seems to surpass the area of COMPRESS-generated designs, albeit by a very small margin at the cost of a really slow COMPRESS (317×). In terms of solver time, TETRIS performs consistently far better, as can be seen from the last column of the table. The overall average speedup across all 36 COMPRESS entries is 42.4×. At latency 4, TETRIS dominates both area and speed. At latency 5, TETRIS has a smaller area compared to Base/Sep optimized COMPRESS designs, and trades 5% area against COMPRESS: Opt for massive speed gains. At latency 6, TETRIS has 6–13% more area compared to Opt designs via COMPRESS but is up to 317× faster. The area penalty TETRIS pays grows with latency, but so does the speed advantage: making it

Table 4: MRLC comparison with COMPRESS: Skinny S-box Results

$d-1$	Latency	Design	#Random	Area (kGE) w/o rand	Area (kGE) w/ rand	Time (sec)	Speedup
1	4	Tetris	12	**0.63**	**1.11**	0.022	—
		Comp: Base		1.01	1.48	0.26	11.8×
		Comp: Sep		0.97	1.45	0.46	20.9×
		Comp: Opt		0.84	1.32	0.55	**25.0**×
2	4	Tetris	36	**1.41**	**2.83**	0.026	—
		Comp: Base		2.1	3.51	0.27	10.4×
		Comp: Sep		2.04	3.46	0.44	16.9×
		Comp: Opt		1.85	3.27	0.59	**22.7**×
3	4	Tetris	72	**2.47**	**5.3**	0.03	—
		Comp: Base		3.57	6.41	0.3	10.0×
		Comp: Sep		3.5	6.34	0.45	15.0×
		Comp: Opt		3.32	6.16	0.59	**19.7**×
4	4	Tetris	120	**3.84**	**8.56**	0.054	—
		Comp: Base		5.44	10.16	0.27	5.0×
		Comp: Sep		5.36	10.08	0.43	8.0×
		Comp: Opt		5.11	9.83	0.59	**10.9**×
1	5	Tetris	9	**0.72**	**1.07**	0.018	—
		Comp: Base		1.14	1.49	0.38	21.1×
		Comp: Sep		1.12	1.47	0.66	36.7×
		Comp: Opt		0.98	1.34	0.83	**46.1**×
2	5	Tetris	27	**1.61**	**2.67**	0.028	—
		Comp: Base		2.36	3.43	0.35	12.5×
		Comp: Sep		2.33	3.39	0.64	22.9×
		Comp: Opt		2.13	3.2	0.88	**31.4**×
3	5	Tetris	54	**2.86**	**4.99**	0.039	—
		Comp: Base		4.03	6.15	0.34	8.7×
		Comp: Sep		3.98	6.11	0.63	16.2×
		Comp: Opt		3.86	5.99	0.86	**22.1**×
4	5	Tetris	90	**4.47**	**8.02**	0.429	—
		Comp: Base		6.13	9.67	0.35	0.8×
		Comp: Sep		6.08	9.62	0.62	1.4×
		Comp: Opt		5.89	9.43	0.84	**2.0**×
1	6	Tetris	8	**0.83**	**1.15**	0.034	—
		Comp: Base		1.25	1.57	0.42	12.4×
		Comp: Sep		1.23	1.55	0.82	24.1×
		Comp: Opt		1.1	1.42	1.21	**35.6**×
2	6	Tetris	24	**1.86**	**2.81**	0.023	—
		Comp: Base		2.56	3.5	0.42	18.3×
		Comp: Sep		2.53	3.47	0.82	35.7×
		Comp: Opt		2.34	3.28	1.2	**52.2**×
3	6	Tetris	48	**3.3**	**5.19**	0.027	—
		Comp: Base		4.32	6.21	0.42	15.6×
		Comp: Sep		4.28	6.17	0.79	29.3×
		Comp: Opt		4.18	6.07	1.24	**45.9**×
4	6	Tetris	80	**5.16**	**8.31**	0.036	—
		Comp: Base		6.54	9.69	0.42	11.7×
		Comp: Sep		6.49	9.63	0.78	21.7×
		Comp: Opt		6.33	9.47	1.13	**31.4**×

a very favorable tradeoff, specifically given that TETRIS does not rely on any specific optimizations like sharing input-masking sub-circuits across different instances of the same gadget in the design, taking the same input as in COMPRESS. With the COMAR gadget in the DSE, the randomness is further optimized as can be seen from the $Tetris : w/comar$ row field for $d-1=1$. In this case, a design containing the COMAR design with latency 4 could not be realized as expected due to the latency of the COMAR gadget being 2.

13.3.2 MRLC on Canright's AES S-box

Table 3 lists the result of our MRLC algorithm's results on the Canrights' S-box for orders $(d-1)$ ranging from 1 to 4 for *target_latency* ranging from 4 (number of levels) to 6. As can be seen, in this case, the usage of the COMAR gadget ($d-1=1$) causes a significant reduction in randomness over the baseline TETRIS algorithm. Also, randomness-reuse via strategies in the Sep optimization of COMPRESS [CGM+24] results in a lower randomness design (Tetris: rand_reuse). Even though COMPRESS [CGM+24] provides results on the AES Canright's S-box, it uses hand-optimized sub-modules with multiple gadget instances, instead of gadgets, to obtain an optimal masked design for the case of AES Canright's S-box, and hence a direct area comparison with our tool is not possible.

13.3.3 MRLC vs COMPRESS on SKINNY S-box

Table 4 lists the result of comparing our MRLC algorithm's results on the SKINNY S-box for orders $(d-1)$ ranging from 1 to 4 for *target_latency* ranging from 4 (number of levels) to 6. TETRIS matches COMPRESS in the number of random bits required for each *target_latency* across all masking orders. TETRIS achieves strictly lesser area (without including area due to random number generation) than all COMPRESS variants across every configuration in this benchmark, with average area savings of 29.5%, 28.9%, and 24.3% at latencies 4, 5, and 6, respectively. Against the most optimized COMPRESS: Opt designs too, TETRIS is 21–25% lesser in area. In terms of solver time in COMPRESS versus DSE time in TETRIS, the overall average speedup with TETRIS across all 36 COMPRESS entries is 20.3×, with speedups increasing at higher latencies, reaching up to 52.2× at latency 6. Overall, for the Skinny S-box, TETRIS performs better than COMPRESS in both area and speed. In the case of skinny, none of the optimized designs required the usage of the COMAR gadget for the given target latency.

13.3.4 MRLC vs SOTA for AES-128 masked using TETRIS-generated Boyar and Peralta's S-box

Our tool currently does not support the optimization and synthesis of a full-cipher design. Thus, we have included full cipher designs by embedding S-boxes found with our tool into an available AES-128 full cipher. The full cipher design thus obtained has a larger area than the full cipher designs generated via COMPRESS. This is because our MRLC optimizations optimize only the S-box area, leaving out vast opportunities for area optimization across rounds and in the linear parts of the cipher. In turn, COMPRESS optimizes the number of pipeline registers throughout the full cipher (Figure 4 in [CGM+24]) and performs register re-use through gadget decomposition (Section 4 in [CGM+24]).
For a fair comparison with COMPRESS, we replaced the AES Boyar and Peralta's S-box generated via COMPRESS into the same AES-128 pipelined full cipher wrapper used for TETRIS. In doing so, we observe that the area gap between TETRIS and COMPRESS is primarily due to full-cipher optimizations in COMPRESS rather than the S-box-level DSE. Table 5 lists the result of comparing our MRLC algorithm's results on the AES-128 with Boyar and Peralta's S-box for orders $(d-1)$ ranging from 1 to 2 for `target_latency` of 4. Table 5 also lists the synthesis results of full AES designs obtained in [KM22,KM23]. [KM22] and [KM23] report results for full AES-128 round-based encryption. Round-based designs reuse a single round of hardware iteratively, optimizing for area at the cost of throughput, whereas TETRIS produces a fully pipelined design that instantiates all rounds with better throughput at higher area. This makes the area values non-comparable to TETRIS. [KM22] uses AGEMA to generate the round-based designs. [CGM+24] provides AGEMA-generated (in pipelined mode) AES-128 designs for the gadget used to mask the designs in [KM22] (HPC3) and hence the AGEMA-HPC3 row has been added for better comparison with [KM22]. In addition, since [CGLS20] does not report full-cipher implementations for AES-128, the AGEMA-HPC2 pipelined row serves as the baseline corresponding to their gadget (HPC2). Notably, TETRIS achieves lower area than the AGEMA-HPC3 pipelined baseline at the same latency and a lower area than AGEMA-HPC2 at a lower latency, demonstrating the benefit of TETRIS's gadget-level DSE over uniform single-gadget assignment [KM22, CGLS20].

13.4 MLRC Results

Tables 6, 7, and 8 present the MLRC results for the Boyar-Peralta, Skinny, and Canright S-boxes, respectively, across security orders 1 to 4 and varying *target_latency* values with the $Gadgets_List = \{HPC1, HPC2, HPC2o, HPC3, HPC3o\}$. Note that COMPRESS does not support MLRC. For each *target_randomness*, the algorithm is able to satisfy it with minimum latency. At lower latencies, the MLRC algorithm can achieve significant reductions in randomness below the target. At higher latencies, we can see a trade-off

Table 5: MRLC comparison with SOTA for TETRIS's Full AES 128 - 16 round pipelined implementation

Design	Datapath width	Latency	Security order	Area (kGE)
AGEMA-HPC2 (pipelined) [KMMS21] (Boyar-Peralta)	128-bit	91	1	170.9†
			2	354.2†
			3	602.9†
			4	917.2†
AGEMA-HPC3 (pipelined) [KMMS21] (Boyar-Peralta)	128-bit	51	1	148.4†
			2	344.2†
			3	621.1†
			4	979.2†
HPC3 (round-based) [KM22]	128-bit	55	1	44.6
			2	108.5
COMAR (round-based) [KM23]	128-bit	473	1	140.2
COMPRESS: Opt [CGM+24] (Canright's)	128-bit	51	1	85.0†
			2	190.9†
COMPRESS: Opt [CGM+24] (Boyar-Peralta)	128-bit	51	1	108.4†
			2	268.3†
COMPRESS: Opt [CGM+24] (Boyar-Peralta, same AES wrapper as TETRIS)	128-bit	51	1	86.3‡
TETRIS (Boyar-Peralta, same AES wrapper as [CGM+24])	128-bit	51	1	96.1‡
			2	321.4‡

†Area w/ PRNG. ‡Area wo/ PRNG.

between area and randomness: allowing more randomness than the minimum (e.g., 26 vs 24 bits for Skinny at order 2, latency 8) yields significantly smaller area, indicating the steeper randomness-area Pareto frontier. The DSE runtimes are an average of 0.22s for Boyar-Peralta, 0.07s for Skinny, and 0.20s for Canright's S-boxes.

Table 6: MLRC Results: Boyar-Peralta's for each Latency and Security Order

Security Order	Latency	Target_Randomness	Randomness_used	Area (kGE) (w/o rand)	Time (sec)
1	4	57	46	2.81	0.167
		58	46	2.81	0.166
	5	43	37	3.35	0.094
		44	37	3.35	0.092
	6	36	36	3.75	0.061
		37	37	3.74	0.064
	7	34	34	4.20	0.058
		35	35	4.16	0.066
2	4	172	138	6.66	0.298
		174	138	6.66	0.292
	5	130	111	7.91	0.159
		132	111	7.91	0.315
	6	108	108	8.75	0.228
		110	110	8.68	0.118
	7	102	102	9.79	0.073
		104	104	9.67	0.078
3	4	344	276	12.13	0.444
		348	276	12.13	0.467
	5	260	222	14.39	0.154
		264	222	14.39	0.154
	6	216	216	15.81	0.123
		220	219	15.66	0.106
	7	204	204	17.67	0.093
		208	207	17.42	0.088
4	4	570	460	18.57	0.879
		575	460	18.57	0.916
	5	430	370	22.03	0.305
		440	370	22.03	0.301
	6	360	360	24.13	0.155
		365	364	23.86	0.130
	7	340	340	27.00	0.144
		345	344	26.66	0.119

13.5 Masking Formal Verification and PINI guarantee

It is necessary to verify that the designs produced by our DSE framework are indeed secure. Towards that, we have provided formal verification results for PINI-masking of the TETRIS-generated designs, and a PINI-guarantee proof for TETRIS.

Table 7: MLRC Results Skinny S-box for each Latency and Security Order

Security Order	Latency	Target_Randomness	Randomness_used	Area (kGE) (w/o rand)	Time (sec)
1	4	21	12	0.59	0.028
		22	12	0.59	0.028
	5	15	9	0.69	0.023
		16	9	0.69	0.023
	6	11	8	0.80	0.018
		12	8	0.80	0.297
	7	9	8	0.91	0.016
		10	8	0.91	0.018
	8	8	8	1.00	0.022
2	4	64	36	1.34	0.041
	5	46	27	1.56	0.030
		48	27	1.56	0.031
	6	34	24	1.80	0.027
		36	24	1.80	0.069
	7	28	24	2.01	0.023
		30	24	2.01	0.024
	8	24	24	2.17	0.021
		26	26	1.99	0.022
3	4	126	72	2.39	0.246
		129	72	2.39	0.252
	5	90	54	2.80	0.036
		96	54	2.80	0.037
	6	66	48	3.20	0.030
		72	48	3.20	0.031
	7	54	48	3.53	0.027
		57	48	3.53	0.028
	8	48	48	3.78	0.026
		51	51	3.40	0.048
4	4	212	120	3.53	0.403
		216	120	3.53	0.417
	5	152	90	4.07	0.047
		160	90	4.07	0.048
	6	112	80	4.65	0.060
		120	80	4.65	0.039
	7	92	80	5.11	0.034
		96	80	5.11	0.032
	8	80	80	5.47	0.032
		84	84	4.83	0.030

13.5.1 Formal Verification using VERICA

We evaluated the masking security of the TETRIS-generated S-box RTLs (after all the Steps in Section 5, which includes the Transforms 1, 2, 3, and 4) using VERICA [RBFSG22] (Version 3.1.0) under the dth PNI, PSNI, and PINI models in the presence of glitches, where d denotes masking order. The designs were synthesized using the NanGate45 standard cell library with Synopsys Design Compiler (in accordance with VERICA [RBS22] requirements). For the 1st- order masked designs generated by TETRIS, VERICA verified the probing, PNI, PSNI, and PINI masking properties within a practical time budget. For the 2nd-order designs, while the TETRIS-generated masked AES Canright's S-box could be verified for PINI, AES Boyar-Peralta's S-box, and the SKINNY S-box designs timed out in VERICA. They could, however, be verified for PROBING security. VERICA could not scale to verify the 3rd and 4th-order masked designs and timed out. Table 9 summarizes the outcome of verifying the higher-order designs using VERICA. The timeouts are consistent with known scalability limitations of BDD-based exhaustive verification at higher masking orders.

Table 9: VERICA Verification Coverage Across Designs and Masking Orders

Design	**Order 1**	**Order 2**	**Order 3**	**Order 4**
Boyar–Peralta (AES)	PINI ✓	PROBING ✓; PINI: timeout	timeout	timeout
Canright (AES)	PINI ✓	PINI ✓	timeout	timeout
SKINNY	PINI ✓	PROBING ✓; PINI: timeout	timeout	timeout

In the next Sub-Section, we provide a proof-by-construction of the PINI guarantee for masked designs generated via TETRIS. That, along with the VERICA verification of the tractable designs, provides confidence in the security guarantees of the TETRIS DSE framework.

Table 8: MLRC Results for Canright's AES S-box: for each Latency and Security Order

Security Order	Latency	Target_Randomness	Randomness_used	Area (kGE) (w/o rand)	Time (sec)
1	4	57	46	2.81	0.192
		58	46	2.81	0.159
	5	43	37	3.35	0.101
		44	37	3.35	0.084
	6	36	36	3.75	0.124
		37	37	3.74	0.067
2	4	172	138	6.66	0.255
		174	138	6.66	0.312
	5	130	111	7.91	0.110
		132	111	7.91	0.110
	6	108	108	8.75	0.077
		110	110	8.68	0.076
3	4	344	276	12.13	0.325
		348	276	12.13	0.393
	5	260	222	14.39	0.141
		264	222	14.39	0.141
	6	216	216	15.81	0.130
		220	219	15.66	0.106
4	4	570	460	18.57	0.637
		576	460	18.57	0.676
	5	432	370	22.03	0.179
		444	370	22.03	0.182
	6	360	360	24.13	0.134
		366	364	23.86	0.147

13.5.2 TETRIS is PINI Composable

In this section, we give a proof-by-construction of the PINI guarantee for the TETRIS framework. To do so, we first state the PINI definition as defined in [CGLS20].

Probe Isolating Non-Interference (PINI): The Probe-Isolating Non-Interference (PINI) notion introduced by Cassiers and Standaert [CGLS20] is a standard for trivial composition of gadgets to form a gadget-based masked hardware. Intuitively, a gadget is said to be $t-$PINI if any set of t probes on its internal *wires* and *output shares* can be simulated using at most t input shares. More formally, a masked design D operating on d shares is $t-$PINI if for any set of t_1 probes on internal wires and t_2 probes on its output shares such that $t_1 + t_2 \leq t$, the probes can be simulated using a set of at most t_1 input shares. [CGLS20] states with *proof* the following propositions related to the construction of a PINI design from PINI gadgets, which we state below:

1. *Any $t-$PINI gadget (with a number of shares $d > t$) is $t-$probing secure.* (*Proposition 2 in [CGLS20]*)

2. *(PINI composability): Any composite gadget made of $t-$PINI composing gadgets is $t-$PINI.* (*Proposition 3 in [CGLS20]*). Here, a composite gadget is a larger design constructed from t-PINI gadgets.

3. *(Trivial Composability of Linear Gadgets) The trivial implementation of a linear function is t-PINI for any t.* (*Proposition 4 in [CGLS20]*)

Based on these three Propositions on PINI composition, we prove the PINI guarantee of TETRIS below.

Lemma 4 (TETRIS is PINI-Composable)**.** *Let D be the masked design output by TETRIS for an unmasked design with gadget-tree T at masking order d. Then D is d-PINI.*

Proof Idea. Given a design D generated via TETRIS, we make and justify the following claims:

1. **Each linear gadget is d-PINI.**

   By construction (Algorithms 1, 2), every XOR or NOT gate, the only linear gates in D are computed sharewise with no refresh exactly as suggested in the *Trivial Composability of Linear Gadgets* Proposition from above.

2. **Each non-linear gadget is d-PINI.**

   Each non-linear gadget available to TETRIS — HPC1, HPC2 [CGLS20], HPC3 [KM22], and the Toffoli variants: HPC2o/HPC3o [CGM+24] are proven d-PINI at arbitrary order in the cited works. COMAR [KM23] is proven to be first-order probing-secure with composability.[2] TETRIS restricts COMAR to first-order designs. Thus, all the non-linear gadgets used by TETRIS are PINI in the given order.

3. **The asymmetric-gadget transforms (Section 9) preserve PINI of each composing gadget.**

   Transforms 1 to 4 (Algorithms 7 and 8) perform either of the two operations on each gadget:

   - *flipping/reversing* the physical inputs connected to an input share (Transform 1).
   - *substituting* a symmetric gadget for an asymmetric one in a fixed configuration (Transforms 2, 3, and 4).

   In both cases, the resulting node in the *Gadget_tree* is still one of the gadgets covered by Claims 1 and 2, flipping changes only which incoming edge is mapped to which input port of the gadget, which is already covered by Claim 2. The PINI property of a gadget, as established in the cited works, does not depend on a fixed labeling of which physical port it is. Hence, every vertex in D, after any sequence of Transforms 1, 2, 3, and 4, remains a d-PINI composing gadget.

4. **D is a valid gadget composition under PINI.**
   By construction (Section 5, Steps 1–4), the gadget-tree G underlying D is a directed acyclic graph whose vertices are either non-linear gadgets (HPC1, HPC2, HPC3, their Toffoli variants, or COMAR), linear/NOT gadgets (Algorithms 1, 2), or primary inputs/outputs, with each composing gadget's m inputs in one-to-one correspondence with its incoming edges. This satisfies the structural requirements of a gadget composition as required by *PINI Composability* Proposition [CS20].

Thus, from Claim 4, we can conclude by construction that a TETRIS-generated design D is PINI-composable [3].

14 Conclusion and Future Work

We have presented two independent formulations which are dual to each other with the objective of automated DSE of gadget-based masked hardware. We have identified an underlying structural parameter for optimizing gadget-based masked hardware: minimizing *balancing registers*. We show that gadget-based masked hardware provides adequate structural properties to allow a faster exploration of the randomness-latency trade-off, rendering expensive, exhaustive global search and optimization unnecessary. The tool has been open-sourced[4]. TETRIS focuses on the DSE of gadget-based masked hardware within non-linear gadgets. The initial pass assigns optimal PINI-composable linear gadgets and does not explore how different linear gadgets can lead to the usage of different non-linear gadgets in composition. Thus, including both linear and non-linear gadgets to explore the trade-off is the next logical extension to this work. With the inclusion of composition laws across various security notions like Non-Interference(NI), Strong Non-Interference(SNI),

[2] The COMAR paper does not formally prove PINI. We have verified via VERICA that COMAR satisfies PINI at first order, both standalone and in composition with HPC and COMAR gadgets.

[3] For designs containing COMAR, PINI has been verified empirically via VERICA, as COMAR [KM23] does explicitly prove PINI composability.

[4] https://github.com/nilotpolas/TETRIS-artifact.

Probe-Isolating Non-Interference(PINI), etc., this problem becomes more interesting, which can be a future work. This will also enable more gadgets in the DSE. We believe that given the vast array of inter-gadget as well as design-dependent optimizations for masked hardware, which a single DSE tool might not be able to cover, TETRIS can provide a good starting point for such a fine-tuned DSE.

References

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 616–648. Springer, 2016.

[Ben12] E Bendersky. Pycparser c parser and ast generator written in python. https://pypi.org/project/pycparser/, 2012.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 326–343. Springer, 2014.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Annual international cryptology conference*, pages 123–153. Springer, 2016.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the aes s-box. In *IFIP international information security conference*, pages 287–298. Springer, 2012.

[Can05] David Canright. A very compact s-box for aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 441–455. Springer, 2005.

[Cas23] Gaëtan Cassiers. Compress. https://github.com/cassiersg/compress, 2023. Accessed: 2026-04-09.

[CG00] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–237. Springer, 2000.

[CGLS20] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Transactions on Computers*, 70(10):1677–1690, 2020.

[CGM+24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Generate small and fast masked pipelined circuits. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):500–529, 2024.

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[CMM+24] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking – unrolled trivium to the rescue. *IACR Communications in Cryptology*, 1(2), 2024.

[CS19] Gaëtan Cassiers and François-Xavier Standaert. Towards globally optimized masking: From low randomness to low noise rate: Or probe isolating multiplications with reduced randomness and security against horizontal attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 162–198, 2019.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[GKM+24] Barbara Gigerl, Franz Klug, Stefan Mangard, Florian Mendel, and Robert Primas. Smooth passage with the guards: Second-order hardware masking of the aes with low randomness and low latency. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):309–335, 2024.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[HB24] Vedad Hadžić and Roderick Bloem. Efficient and composable masked aes s-box designs using optimized inverters. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):656–683, Dec. 2024.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[IYL+23] Saya Inagaki, Mingyu Yang, Yang Li, Kazuo Sakiyama, and Yuko Hara-Azumi. Power side-channel attack resistant circuit designs of arx ciphers using high-level synthesis. *ACM Transactions on Embedded Computing Systems*, 22(5):1–17, 2023.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1799–1812, 2022.

[KM23] David Knichel and Amir Moradi. Composable gadgets with reused fresh masks − first-order probing-secure hardware circuits with only 6 fresh masks. *Cryptology ePrint Archive*, 2023.

[KMMS21] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *Cryptology ePrint Archive*, 2021.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits: Towards automated generation of composable secure gadgets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 323–344, 2022.

[Mic94] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited: Or why proofs in the robust probing model are needed. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 256–292, 2019.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M Gammel. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[RBFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. Verica-verification of combined attacks: Automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 255–284, 2022.

[RBS22] Jan Richter-Brockmann and Pascal Sasdrichs. Verica. https://github.com/Chair-for-Security-Engineering/VERICA, 2022. Accessed: 2026-06-15.

[SSM21] Rajat Sadhukhan, Sayandeep Saha, and Debdeep Mukhopadhyay. Shortest path to secured hardware: Domain oriented masking with high-level-synthesis. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 223–228. IEEE, 2021.

[STK24] Nilotpola Sarma, Anuj Singh Thakur, and Chandan Karfa. Maskedhls: Domain-specific high-level synthesis of masked cryptographic designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3973–3984, 2024.

[TY15] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.