IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 656–683. DOI:10.46586/tches.v2025.i1.656-683

Efficient and Composable Masked AES S-Box Designs Using Optimized Inverters

Vedad Hadžić and Roderick Bloem

Graz University of Technology, Graz, Austria, first.last@iaik.tugraz.at

Abstract. Hardware implementations of cryptographic algorithms are susceptible to power analysis attacks, allowing attackers to break the otherwise strong security guarantees. A theoretically sound countermeasure against such attacks is masking, where all key- and data-dependent intermediate values in the computation are split into so-called shares, requiring an attacker to learn all of them before recovering the secret key. Masking a cryptographic hardware design against power analysis attacks incurs large area and latency overheads due to their nonlinear components, especially when implemented using composable masking schemes.

These overheads disproportionately affect ciphers with highly nonlinear monolithic S-Boxes like the Advanced Encryption Standard (AES). The masking of the AES S-Box is well studied, and most implementations use Canright's $\mathbb{F}_{2^8}$ inverter design that decomposes operations in a larger field into a combination of multiplications, additions and inversions in a smaller field. While remarkable, Canright's inverter design has a sub-optimal multiplicative depth, and can thus not take full advantage of recent developments in low-latency composable masking schemes.

In this paper, we present a $\mathbb{F}_{2^8}$ inverter that achieves the optimal multiplicative depth of *three*, and use it to construct a more efficient trivially composable masked implementation of the AES S-Box. Moreover, we present HPC3.1, a better low-latency multiplication gadget that works in all finite fields $\mathbb{F}_{p^n}$, and a randomness reuse strategy for both HPC1 and HPC3.1 gadgets that preserves side-channel security. Orthogonally, we also propose an improved bit-level implementation of the $\mathbb{F}_{2^4}$ inverter for more efficient masked S-Box designs based on Canright's original $\mathbb{F}_{2^8}$ inverter.

We develop, functionally test, and formally verify the trivially composable side-channel security of all masked AES S-Box designs. Our evaluation shows that the designs outperform or match the state-of-the-art in terms of latency, randomness use and area cost.

Keywords: AES, Masking, PINI, Low-latency, Mask Reuse

1 Introduction

All electronic devices consume power while performing computations, and their power consumption depends on the data that is being processed. It turns out that the side-channel information revealed through power consumption is sufficient to mount powerful attacks against otherwise secure cryptographic algorithms. Differential power analysis (DPA) [KJJ99] and correlation power analysis (CPA) [BCO04] are examples of such attacks, where an attacker observes the power consumption for different controllable inputs, creates predictions for the power consumption of secret key candidates, and finally scores the key candidates based on how well the prediction correlates with real power measurements.

Masking is an algorithmic countermeasure against both DPA and CPA attacks, that transforms the original computation so that every secret, data and intermediate value is split into several random and statistically independent components called *shares*. The

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

purpose of this change is clear: make it so that an attacker needs to combine information about all shares in order to recover the secret, forcing them to perform either multivariate correlation analysis or correlate on higher-order statistical moments, hopefully achieving a security gain exponential in the number of shares [CJRR99].

There have been several proposals for masking schemes that achieve security against DPA and CPA attacks. These include threshold implementations (TI) [NRS11], domain-oriented masking (DOM) [GMK16], the consolidated masking scheme (CMS) [RBN+15], generic low-latency masking (GLLM) [GIB18], and more recently hardware-private circuits (HPC) [CGLS21]. In general, these masking approaches significantly increase the size of the circuit, introduce additional latency to protect against glitches, and require fresh uniformly random values when masking nonlinear functions.

This masking overhead is particularly noticeable in the highly nonlinear S-Box of AES [DR98]. It has long since been the prime target for various masked hardware designs [MPL+11, CBR+15, BGN+15, CRB+16, GC17, UHA17, GIB18, SGMT18, GSM+19, SBHM20, ADN+22]. Most of these masked implementations are, either directly or indirectly, based on Canright's $\mathbb{F}_{2^8}$ inverter [Can05], which recursively implements operations in a larger field from operations in a smaller component field, all the way to $\mathbb{F}_2$ where additions, multiplications and inverses become Xor gates, And gates and identity operations.

In this paper, we present a novel approach to the problem of efficiently masking the AES S-Box. Our goal is to design a masked AES S-Box which fulfills trivial composability through *probe-isolating non-interference* (PINI) and has low latency, chip area, and randomness requirements. We achieve this objective through several contributions:

(1) **An improved $\mathbb{F}_{2^8}$ inverter design**. We present the first small $\mathbb{F}_{2^8}$ inverter design that achieves the optimal multiplicative depth of *three* instead of *four*, enabling efficient HPC masking of the AES S-Box with three clock cycles of latency.
(2) **A better masked multiplier in $\mathbb{F}_q$.** We present HPC3.1, a better masked multiplication gadget that works in arbitrary fields $\mathbb{F}_q$ and requires less operations than its predecessor, while keeping its low latency and composability properties.
(3) **A sound randomness reuse strategy.** We present a randomness reuse strategy that enables randomness sharing between gadgets while preserving secure composability. This significantly reduces the randomness cost of masked AES S-Boxes.
(4) **Better masked AES S-Box designs.** We present several trivially composable masked AES S-Box designs that match or outperform the state of the art in all relevant metrics. This includes the first ever trivially composable design that achieves a latency of three clock cycles at arbitrary protection orders, as well as the most efficient trivially composable masked AES S-Box based on Canright's $\mathbb{F}_{2^8}$ inverter.

Outline. This paper follows a natural structure, where Section 2 presents background on side-channel formalism, PINI theory, HPC gadgets, and Canright's $\mathbb{F}_{2^8}$ inverter, Section 3 presents our new $\mathbb{F}_{2^8}$ inverter, followed by Sections 4 and 5 which introduce the HPC3.1 gadget and show how randomness can be reused across gadgets. Afterwards, Section 6 presents candidate masked $\mathbb{F}_{2^8}$ inverter designs and Section 7 evaluates masked AES S-Boxes based on said designs, comparing them to related work. Section 8 concludes the paper. Appendices A and B show an additional gadget and unmasked bit-level results.

2 Background

In this section, we briefly revisit the background and notation relating to formal models of power side-channel attacks. Afterwards, we present probe-isolating non-interference as a road to composable side-channel security and present two gadgets that achieve this strict security notion. Finally, we give an overview of Canright's $\mathbb{F}_{2^8}$ decomposition employed in efficient hardware implementations of the AES S-Box.

2.1 Power Side Channels and Masking

The power consumption of real-world devices is all but simple, and many papers have looked at ways to define a formal framework for attacks like DPA and CPA that exploit this side channel. One such model is the so-called *probing model* [ISW03], where an attacker can pick up to d intermediate values in a computation and directly read their values. The security order $d \geq 1$ roughly corresponds to the DPA and CPA attack order, *i.e.,* either the statistical moment of power consumption correlated in the attacks or the number of timepoints combined in a multivariate attack. Either way, we say that a computation is d probing secure if an attacker able to repeatedly probe (read) any d intermediate values in the computation cannot recover any information about the secrets.

Since the original introduction of the probing model, follow-up work has determined that the definition of intermediate values in a hardware execution is not straightforward. This is primarily due to *glitches*, *i.e.,* transient computations caused by value changes and signal timings that influence the power consumption but not the synchronous state of a hardware circuit. Adding these transient intermediate computations to the probing model yields the so-called *glitch-extended probing model* [FGP+18]. Take for example the computation $C = A + B$. In hardware, depending on the exact timing, the signal C may first take on the value of either A or B before the other "arrives" and C stabilizes to the actual value $A + B$. One could say that an adversary placing a probe on C observes the set of values $\rho(C) = \{A, B\}$ in the worst case. These glitch extended probes are very powerful and give an attacker probing C access to all values that drive it, *i.e.,* the set of all registers and inputs at the base of the computational cone of C. Registers stop the propagation of glitches because they only change their value when the clock ticks, e.g., on a positive clock edge. In the rest of this paper, we are always considering glitch-extended probes.

Intuitively, an attacker able to read any value in the computation is able to break straightforward cryptographic algorithm implementations, so more sophisticated implementation countermeasures are necessary. One such countermeasure is *masking*. Masking is a secret sharing technique where all inputs, intermediates and outputs of a computation are split into sets of values called *shares*, with the idea that an attacker needs knowledge of all shares to recover the respective original value. Let $\mathbf{X}_* = \{X_0, \ldots, X_{n-1}\}$ be the set of all original computation inputs and $\mathbf{Y}_* = \{Y_0, \ldots, Y_{m-1}\}$ be the set of all original computation outputs. In additive masking over a finite field $\mathbb{F}_q$, every secret input X_k is split into a set of *shares* $\mathbf{X}_{k,*} = \{X_{k,0}, \ldots, X_{k,d}\}$, such that $X_k = \sum_{i=0}^{d} X_{k,i}$. The shares $X_{k,i}$ are supposed to be independent and uniformly random so that the knowledge of up to d shares does not reveal any information about X_k, *i.e.,* without knowledge of the last missing share, each original value of X_k is equally likely. We additionally use the notation $\mathbf{X}_{*,i} = \{X_{0,i}, \ldots, X_{n-1,i}\}$ for the i^{th} share of each masked input, and $\mathbf{X}_{*,*} = \bigcup_{k=0}^{n-1} \mathbf{X}_{k,*} = \bigcup_{i=0}^{d} \mathbf{X}_{*,i}$ for the set of all shares of all inputs. Similarly, the original computation outputs Y_k are also split into the shares $\mathbf{Y}_{k,*} = \{Y_{k,0}, \ldots, Y_{k,d}\}$, and we write $\mathbf{Y}_{*,i} = \{Y_{0,i}, \ldots, Y_{m-1,i}\}$ for the i^{th} share of each masked output, and $\mathbf{Y}_{*,*} = \bigcup_{k=0}^{m-1} \mathbf{Y}_{k,*} = \bigcup_{i=0}^{d} \mathbf{Y}_{*,i}$ for all shares of all masked outputs. When adapting a computation to a masked computation, it is often necessary to sample new uniformly random values on the fly to *reshare* sharings of intermediate variables or *blind* sharings during certain operations. Here, we think of these uniform random values as additional inputs $\mathbf{R}$ to the masked computation. The original computation is adapted to these shared signals. For simplicity, we write $\mathbf{T}$ for the set of all intermediate values of the masked computation. For the probing model, these intermediate computations are a special kind of output that an attacker can get access to, and Definition 1 formalizes this intuition.

Definition 1 (Masked Computation)**.** A masked computation is a function $\psi : \langle \mathbf{X}_{*,*}, \mathbf{R} \rangle \mapsto \langle \mathbf{Y}_{*,*}, \mathbf{T} \rangle$ mapping a set $\mathbf{X}_{*,*}$ of n shared inputs $\mathbf{X}_*$ and a set $\mathbf{R}$ of uniformly random values to the set $\mathbf{Y}_{*,*}$ of m shared output values $\mathbf{Y}_*$ and a set $\mathbf{T}$ of intermediate values.

Algorithm 1: Generic multiplication gadget HPC1 [CGLS21]

```
Input  : Sharing $\langle A_0, \dots, A_d \rangle \in \mathbb{F}_q^{d+1}$ of $A \in \mathbb{F}_q$,
         sharing $\langle B_0, \dots, B_d \rangle \in \mathbb{F}_q^{d+1}$ of $B \in \mathbb{F}_q$,
         sharing $\langle R_0, \dots, R_d \rangle \in \mathbb{F}_q^{d+1}$ of $0 \in \mathbb{F}_q$,
         masks $\langle P_{i,j} \mid 0 \leq i < j \leq d \rangle \in \mathbb{F}_q^{d(d+1)/2}$
Output : Sharing $\langle C_0, \dots, C_d \rangle \in \mathbb{F}_q^{d+1}$ of $C = A \cdot B$
1 for $i$ from 0 to $d$ do
2     $B_i' \leftarrow \mathrm{Reg}\left(B_i + R_i\right)$;                 // Refresh sharing of $B$
3 for $i$ from 0 to $d$ do
4     $V_{i,i} \leftarrow A_i \cdot B_i'$;                              // Same-domain terms
5     for $j$ from $i+1$ to $d$ do
6         $V_{i,j} \leftarrow P_{i,j} + A_i \cdot B_j'$;               // Cross-domain terms
7         $V_{j,i} \leftarrow -P_{i,j} + A_j \cdot B_i'$;              // Cross-domain terms
8     $C_i = \sum_{j=0}^{d} \mathrm{Reg}\left(V_{i,j}\right)$;
9 return $\langle C_0, \dots, C_d \rangle$;
```

2.2 Probe-Isolating Non-Interference

While probing security gives a good reflection of security against DPA and CPA attacks, it is a monolithic property and does not compose in general. One can have two probing secure computations that, when put together, suddenly become insecure. For this reason, a lot of research went into establishing composable security definitions that imply d-probing security, e.g., strong non-interference (SNI) [BBD+16, BBP+16]. In the following, we give an overview of d-probe-isolating non-interference (d-PINI) [CS20, CGLS21], which is a property that allows for trivial composition of small masked d-PINI computations called *gadgets* into larger masked d-PINI computations.

Like other composable security notions for masked implementations, PINI builds on the concept of probe simulations. That is, proving a computation is d-PINI involves proving that the distribution witnessed by an attacker probing a set of at most d probes on intermediate values and output share domains can be *simulated* while only knowing a set of d input share domains. In the following, we give a definition of simulation and PINI based on conditional probability distributions [KSM20] instead of the more traditional exposition that uses a randomized simulation algorithm [CS20].

Definition 2 (Simulation)**.** Let $\mathbf{A}$ and $\mathbf{B}$ be sets of random variables and let $\mathbf{B}_0 \subseteq \mathbf{B}$ and $\mathbf{B}_1 = \mathbf{B} \setminus \mathbf{B}_0$ partition $\mathbf{B}$. We say that $\mathbf{B}_0$ perfectly simulates the observations of $\mathbf{A}$ under $\mathbf{B}$ if and only if

$$\forall \mathbf{a}, \mathbf{b}_0, \mathbf{b}_1 : \Pr\left[\mathbf{A} = \mathbf{a} \mid \mathbf{B}_0 = \mathbf{b}_0\right] = \Pr\left[\mathbf{A} = \mathbf{a} \mid \mathbf{B}_0 = \mathbf{b}_0, \mathbf{B}_1 = \mathbf{b}_1\right]. \quad (1)$$

Definition 3 (Probe-isolating Non-interference (PINI) [KSM20])**.** Let $\psi : \langle \mathbf{X}_{*,*}, \mathbf{R} \rangle \mapsto \langle \mathbf{Y}_{*,*}, \mathbf{T} \rangle$ be a masked computation. For a set of indices I, let $\mathbf{X}_{*,I} = \bigcup_{i \in I} \mathbf{X}_{*,i}$ be a set of input shares with indices I. The masked computation ψ is d-PINI if for all output share domain probes $\mathbf{Q}_{\text{out}} = \{\rho\left(Y_{k,i}\right) \mid k \in [0, m), i \in I\}$ and internal probes $\mathbf{Q}_{\text{int}} \subseteq \mathbf{T} \cup \mathbf{R} \cup \mathbf{X}_{*,*}$, with $|I| + |\mathbf{Q}_{\text{int}}| \leq d$, there exists a set of indices I', with $|I'| \leq |I| + |\mathbf{Q}_{\text{int}}|$ and $I' \supseteq I$, so that $\mathbf{Q}_{\text{out}} \cup \mathbf{Q}_{\text{int}}$ is perfectly simulated by $\mathbf{X}_{*,I'}$ under $\mathbf{X}_{*,*}$.

Cassiers *et al.* [CGLS21] have adapted the d-PINI security notion to the glitch-extended probing model and presented the first two *hardware private circuit* multiplication gadgets fulfilling the glitch-extended d-PINI property and named them HPC1 and HPC2. The HPC1 gadget is shown in Algorithm 1, and represents an adapted version of the DOM gadget [GMK17] where one of the inputs is reshared. Here, input $\mathbf{R}$ represents a sharing

Algorithm 2: $\mathbb{F}_2$ multiplication gadget HPC3 [KM22]

```
   Input  : Sharing $\langle A_0, \dots, A_d \rangle \in \mathbb{F}_2^{d+1}$ of $A \in \mathbb{F}_2$,
            sharing $\langle B_0, \dots, B_d \rangle \in \mathbb{F}_2^{d+1}$ of $B \in \mathbb{F}_2$,
            masks $\langle R_{i,j} \mid 0 \leq i < j \leq d \rangle \in \mathbb{F}_2^{d(d+1)/2}$,
            masks $\langle P_{i,j} \mid 0 \leq i < j \leq d \rangle \in \mathbb{F}_2^{d(d+1)/2}$
   Output : Sharing $\langle C_0, \dots, C_d \rangle \in \mathbb{F}_2^{d+1}$ of $C = A \cdot B$
1  for $i$ from $0$ to $d-1$ do
2      for $j$ from $i+1$ to $d$ do
3          $R_{j,i} \leftarrow R_{i,j}, P_{j,i} \leftarrow P_{i,j}$;
4  for $i$ from $0$ to $d$ do
5      $U_{i,i} \leftarrow \mathrm{Reg}\,(A_i \wedge B_i)$;
6      for $j$ from $0$ to $d$ with $j \neq i$ do
7          $V_{i,j} \leftarrow R_{i,j} \oplus B_j$; $W_{i,j} \leftarrow P_{i,j} \oplus \neg A_i \wedge R_{i,j}$;
8          $U_{i,j} \leftarrow \mathrm{Reg}\,(A_i) \wedge \mathrm{Reg}\,(V_{i,j}) \oplus \mathrm{Reg}\,(W_{i,j})$;
9      $C_i = \bigoplus_{j=0}^{d} U_{i,j}$;
10 return $\langle C_0, \dots, C_d \rangle$;
```

of $0 \in \mathbb{F}_q$ that is used to reshare $\mathbf{B}$ into $\mathbf{B}'$. Using $\mathbf{B}'$ instead of $\mathbf{B}$ in the rest of the gadget makes all internal probes on $V_{i,j}$ independent of $\mathbf{B}$, even in the presence of glitches, ensuring that A_i simulates $V_{i,j}$ under $\mathbf{A} \cup \mathbf{B}$. However, this comes at the cost of adding an additional clock cycle of latency for the output compared to input sharing $\mathbf{B}$, ultimately leading to an asymmetric output latency of one clock cycle compared to $\mathbf{A}$ and two clock cycles compared to $\mathbf{B}$. The HPC1 gadget has a randomness cost of $d\,(d+1)\,/2 + r\,(d)$ field elements in $\mathbb{F}_q$, where $r\,(d)$ is the number of random bits needed to create the sharing $\mathbf{R}$ of $0 \in \mathbb{F}_2$, e.g., $r\,(1) = 1, r\,(2) = 2, r\,(3) = 4, r\,(4) = 5$ as shown by Cassiers *et al.* [CGLS21].

The HPC2 gadget is a straightforward adaptation of the PINI1 gadget [CS20]. It uses the so-called multiplication trick to ensure correctness and the d-PINI property, which limits it to only work in $\mathbb{F}_2$ unlike HPC1. However, it does achieve the same latency characteristics while requiring only $d\,(d+1)\,/2$ random bits, making it an attractive alternative for masked bit-level computations.

A recent paper by Knichel and Moradi [KM22] proposes the low-latency HPC3 gadget show in Algorithm 2. It performs a d-PINI secure masked multiplication in $\mathbb{F}_2$ and only requires a single register stage, *i.e.,* a latency of one clock cycle compared to both input sharings. This is a significant improvement over HPC1 and HPC2 but comes at the cost of an increased randomness requirement of $d\,(d+1)$ random bits. These additional random bits are necessary for the blind-then-correct approach used in HPC3, where B_j is blinded with $R_{i,j}$ to make the multiplication $A_i \wedge (B_j \oplus R_{i,j})$ independent of the sharing $\mathbf{B}$. Subsequently, the byproduct term $A_i \wedge R_{i,j}$ must be removed through cancellation, which requires blinding it with $P_{i,j}$ and adding it to the result of the previous multiplication.

2.3 Normal Basis Field Decomposition of $\mathbb{F}_{2^8}$

The AES S-Box is the part of AES providing *confusion*, and is defined as a bijective function with 8-bit inputs and outputs. It consists of an inversion in $\mathbb{F}_{2^8}$ followed by an affine transformation. While the affine transformation is easy to implement in hardware, the $\mathbb{F}_{2^8}$ inversion is a highly nonlinear function, where all individual output bits have an algebraic degree of seven. In the following, we give an overview over the mathematical trickery enabling efficient $\mathbb{F}_{2^8}$ inversion, as presented by Canright [Can05].

It turns out that it is possible to represent an element of $\mathbb{F}_{2^{2n}}$ and all $\mathbb{F}_{2^{2n}}$ field operations using two elements of $\mathbb{F}_{2^n}$ and $\mathbb{F}_{2^n}$ field operations. In the following exposition, we use uppercase latin for elements of $\mathbb{F}_{2^8}$, uppercase greek for elements of $\mathbb{F}_{2^4}$, lowercase greek for elements of $\mathbb{F}_{2^2}$ and lowercase latin for elements of $\mathbb{F}_2$.

A general element G of $\mathbb{F}_{2^8}$ can be represented as $G = \Gamma_1 Y^{16} + \Gamma_0 Y$ where Y^{16} and Y are the roots of the irreducible polynomial $q'(y) = y^2 + \Pi y + \Sigma$, with $\Pi = Y^{16} + Y$ and $\Sigma = Y^{16}Y$. Similarly, an element Γ of $\mathbb{F}_{2^4}$ is represented as $\Gamma = \gamma_1 Z^4 + \gamma_0 Z$ using the roots Z^4 and Z of the irreducible polynomial $q''(z) = z^2 + \pi z + \sigma$, with $\pi = Z^4 + Z$ and $\sigma = Z^4 Z$. Finally, an element γ of $\mathbb{F}_{2^2}$ is represented as $\gamma = g_1 W^2 + g_0 W$ using the roots W^2 and W of the only irreducible quadratic $\mathbb{F}_2$ polynomial $q'''(w) = w^2 + w + 1$, with both $W^2 + W = 1$ and $W^2 W = 1$. This decomposition allows for efficient implementation of multiplications and inversions in larger finite fields using simple operations in its component fields. For example, the multiplication of two $\mathbb{F}_{2^8}$ elements $G = \Gamma_1 Y^{16} + \Gamma_0 Y$ and $F = \Phi_1 Y^{16} + \Phi_0 Y$ breaks down to addition, multiplication and scaling by constants in $\mathbb{F}_{2^4}$ where

$$GF = \left(\Gamma_1 Y^{16} + \Gamma_0 Y\right)\left(\Phi_1 Y^{16} + \Phi_0 Y\right) = \Lambda_1 Y^{16} + \Lambda_0 Y = L,$$
$$\text{with} \quad \begin{array}{l} \Lambda_1 = \Gamma_1 \Phi_1 \Pi + (\Gamma_1 + \Gamma_0)(\Phi_1 + \Phi_0)\Sigma\Pi^{-1} \\ \Lambda_0 = \Gamma_0 \Phi_0 \Pi + (\Gamma_1 + \Gamma_0)(\Phi_1 + \Phi_0)\Sigma\Pi^{-1} \end{array} . \tag{2}$$

The same is true for $\mathbb{F}_{2^8}$ inversions $L = G^{-1}$, where

$$G^{-1} = \left(\Gamma_1 Y^{16} + \Gamma_0 Y\right)^{-1} = \Lambda_1 Y^{16} + \Lambda_0 Y = L,$$
$$\text{with} \quad \begin{array}{l} \Lambda_1 = \Gamma_0 \left(\Gamma_1 \Gamma_0 \Pi^2 + (\Gamma_1 + \Gamma_0)^2 \Sigma\right)^{-1} \\ \Lambda_0 = \Gamma_1 \left(\Gamma_1 \Gamma_0 \Pi^2 + (\Gamma_1 + \Gamma_0)^2 \Sigma\right)^{-1} \end{array} . \tag{3}$$

Multiplication and inversion in $\mathbb{F}_{2^4}$ follows the same pattern as in $\mathbb{F}_{2^8}$. Here, the basis $\langle Z^4, Z\rangle$ is used, and all $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^4}$ symbols G, F, L, Γ_i, Φ_i, Λ_i, Π, and Σ are substituted with their respective $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^2}$ counterparts Γ, Φ, Λ, γ_i, ϕ_i, λ_i, π, and σ:

$$\Gamma\Phi = \left(\gamma_1 Z^4 + \gamma_0 Z\right)\left(\phi_1 Z^4 + \phi_0 Z\right) = \lambda_1 Z^4 + \lambda_0 Z = \Lambda,$$
$$\text{with} \quad \begin{array}{l} \lambda_1 = \gamma_1 \phi_1 \pi + (\gamma_1 + \gamma_0)(\phi_1 + \phi_0)\sigma\pi^{-1} \\ \lambda_0 = \gamma_0 \phi_0 \pi + (\gamma_1 + \gamma_0)(\phi_1 + \phi_0)\sigma\pi^{-1} \end{array} , \quad \text{and} \tag{4}$$

$$\Gamma^{-1} = \left(\gamma_1 Z^4 + \gamma_0 Z\right)^{-1} = \lambda_1 Z^4 + \lambda_0 Z = \Lambda,$$
$$\text{with} \quad \begin{array}{l} \lambda_1 = \gamma_0 \left(\gamma_1 \gamma_0 \pi^2 + (\gamma_1 + \gamma_0)^2 \sigma\right)^{-1} \\ \lambda_0 = \gamma_1 \left(\gamma_1 \gamma_0 \pi^2 + (\gamma_1 + \gamma_0)^2 \sigma\right)^{-1} \end{array} . \tag{5}$$

Similarly, multiplication and inversion in $\mathbb{F}_{2^2}$ also follow the same structure with the appropriate basis $\langle W^2, W\rangle$. Here, the symbols G, F, L, Γ_i, Φ_i, Λ_i, Π, and Σ are replaced by their appropriate $\mathbb{F}_{2^2}$ and $\mathbb{F}_2$ counterparts γ, ϕ, λ, g_i, f_i, l_i, $p = 1$, and $s = 1$. Moreover, most of the terms in the $\mathbb{F}_{2^2}$ inversion cancel out, yielding a simple swap of the two component $\mathbb{F}_2$ elements:

$$\gamma\phi = \left(g_1 W^2 + g_0 W\right)\left(f_1 W^2 + f_0 W\right) = l_1 W^2 + l_0 W = \lambda,$$
$$\text{with} \quad \begin{array}{l} l_1 = g_1 f_1 + (g_1 + g_0)(f_1 + f_0) \\ l_0 = g_0 f_0 + (g_1 + g_0)(f_1 + f_0) \end{array} , \quad \text{and} \tag{6}$$

$$\gamma^{-1} = (g_1 W^2 + g_0 W)^{-1} = g_0 W^2 + g_1 W. \tag{7}$$

This "free" $\mathbb{F}_{2^2}$ inversion in the normal basis representation is one of the reasons for preferring it over the more commonplace polynomial basis representation when constructing hardware circuits implementing the AES S-Box.

Figure 1 gives a structural overview of Canright's $\mathbb{F}_{2^8}$ inversion. The only nonlinear operations are the three $\mathbb{F}_{2^4}$ multiplications and the single $\mathbb{F}_{2^4}$ inversion. Addition in $\mathbb{F}_{2^4}$, scaling by $\mathbb{F}_{2^4}$ constants and squaring of $\mathbb{F}_{2^4}$ elements are linear operations.

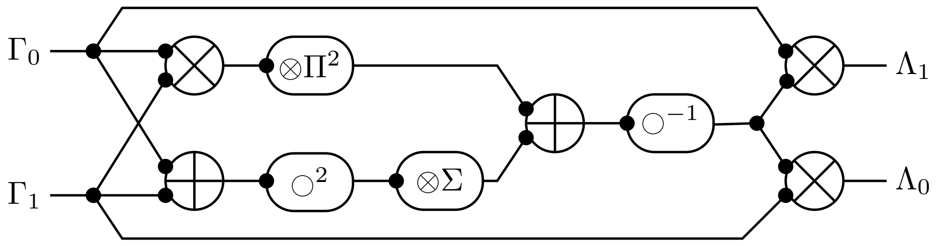


Figure 1: Structure of the $\mathbb{F}_{2^8}$ inversion outlined in equation (3). All signals are $\mathbb{F}_{2^4}$ elements and all operations are in $\mathbb{F}_{2^4}$. Here, $\otimes$ is multiplication, $\oplus$ is addition, $\bigcirc^2$ and $\bigcirc^{-1}$ are squaring and inversion, and $\otimes\_$ scales by the given constant.

3 A Shallower $\mathbb{F}_{2^8}$ Inversion

The $\mathbb{F}_{2^8}$ inversion, interpreted as a polynomial over $\mathbb{F}_2$ has an algebraic degree of $\deg\left(G^{-1}\right) = 7$. Therefore, it must be possible to implement it with a multiplicative depth of at least $\operatorname{dep}\left(G^{-1}\right) \geq \left\lceil \log_2\left(\deg\left(G^{-1}\right)\right)\right\rceil = 3$. An example of a (bad) implementation that achieves $\operatorname{dep}\left(G^{-1}\right) = 3$ is its algebraic normal form where each monomial is implemented as a balanced multiplication tree. The construction given by Canright [Can05] and shown in Figure 1 achieves a multiplicative depth of $\operatorname{dep}\left(G^{-1}\right) \geq 4$. This can be computed from the optimal multiplicative depths of the functions used in its computation. Here, the $\mathbb{F}_{2^4}$ multiplication $\Gamma\Phi$ has degree $\deg(\Gamma\Phi) = 2$ and thus optimal multiplicative depth of $\operatorname{dep}(\Gamma\Phi) \geq 1$, whereas the $\mathbb{F}_{2^4}$ inversion Γ^{-1} has algebraic degree $\deg\left(\Gamma^{-1}\right) = 3$ and thus an optimal multiplicative depth of $\operatorname{dep}\left(\Gamma^{-1}\right) \geq 2$. All other functions are linear, and thus have a degree of $\deg(\cdot) = 1$ and multiplicative depth $\operatorname{dep}(\cdot) = 0$.

In the following, we modify the structure of the $\mathbb{F}_{2^8}$ inversion to achieve the optimal multiplicative depth of $\operatorname{dep}\left(G^{-1}\right) = 3$ without significantly increasing the size of the computation or number of field multiplications. Starting from the structure shown in Figure 1, we can perform a cut in the computation graph such that the inversion in $\mathbb{F}_{2^4}$ and the two final $\mathbb{F}_{2^4}$ multiplications are on the right side, and everything else is on the left side of the cut. We observe that the computation of an inverse, followed by a multiplication has algebraic degree $\deg\left(\Gamma^{-1}\Phi\right) = 4$, but is implemented with a multiplicative depth

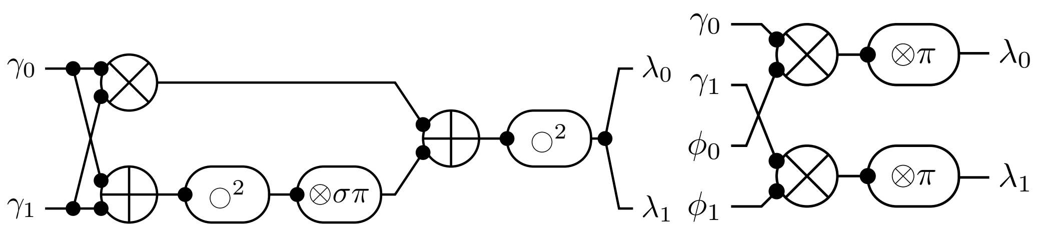
$$\operatorname{dep}\left(\Gamma^{-1}\Phi\right) = \max\left(\operatorname{dep}\left(\Gamma^{-1}\right), \operatorname{dep}(\Phi)\right) + 1 = \max(2, 0) + 1 = 3, \tag{8}$$

which is sub-optimal as $\operatorname{dep}\left(\Gamma^{-1}\Phi\right) \geq \left\lceil \log_2\left(\deg\left(\Gamma^{-1}\Phi\right)\right)\right\rceil = 2$.

Let $\Gamma = \gamma_1 Z^4 + \gamma_0 Z$ and $\Phi = \phi_1 Z^4 + \phi_0 Z$ be elements of $\mathbb{F}_{2^4}$, and let $\Gamma^{-1} = \Lambda = \lambda_1 Z^4 + \lambda_0 Z$ be the inverse of Γ with $\lambda_1 = \gamma_0\theta$, $\lambda_0 = \gamma_1\theta$ and $\theta = \left(\gamma_1\gamma_0\pi^2 + (\gamma_1+\gamma_0)^2\sigma\right)^{-1}$ as in equation (5). The term $\Gamma^{-1}\Phi$ can then be expressed as

$$\begin{aligned}\Gamma^{-1}\Phi = \Lambda\Phi &= (\lambda_1 Z^4 + \lambda_0 Z)(\phi_1 Z^4 + \phi_0 Z) = \\ &= \lambda_1\phi_1\pi Z^4 + \lambda_0\phi_0\pi Z + (\lambda_1+\lambda_0)(\phi_1+\phi_0)\sigma\pi^{-1}(Z^4+Z) = \\ &= \gamma_0\theta\phi_1\pi Z^4 + \gamma_1\theta\phi_0\pi Z + (\gamma_0\theta+\gamma_1\theta)(\phi_1+\phi_0)\sigma\pi^{-1}(Z^4+Z) = \\ &= \theta\left(\gamma_0\phi_1\pi Z^4 + \gamma_1\phi_0\pi Z + (\gamma_0+\gamma_1)(\phi_1+\phi_0)\sigma\pi^{-1}(Z^4+Z)\right).\end{aligned} \tag{9}$$

After factoring out the common term θ, we have balanced the computation in terms of multiplicative depth. Here, $\operatorname{dep}(\theta) = 1$ because the inverse in $\mathbb{F}_{2^2}$ is linear and a $\mathbb{F}_{2^2}$ multiplication has a depth of one, *i.e.,* $\operatorname{dep}(\gamma_1\gamma_0) = 1$. Similarly, the term multiplied with θ has multiplicative depth $\operatorname{dep}(\gamma_0\phi_1) = \operatorname{dep}(\gamma_1\phi_0) = \operatorname{dep}((\gamma_0+\gamma_1)(\phi_1+\phi_0)) = 1$. Furthermore, it is possible to contextualize the obtained result in terms of $\mathbb{F}_{2^4}$ multiplications. For example, the terms multiplied with θ represent a $\mathbb{F}_{2^4}$ multiplication between



(a) Structure of $\Theta(\cdot)$ **(b)** Structure of $\odot$

Figure 2: Structure of the $\Theta(\cdot)$ and $\odot$ operations. Signals and operations are in $\mathbb{F}_{2^2}$, where $\otimes$ is multiplication, $\oplus$ is addition, $\bigcirc^2$ is squaring and $\otimes\_$ scales by the given constant.

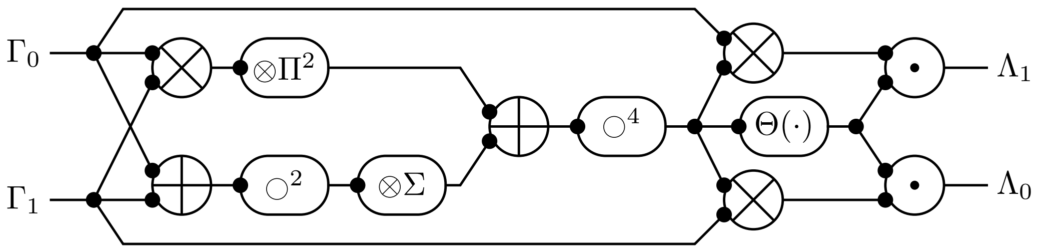


Figure 3: Structure of the new $\mathbb{F}_{2^8}$ inversion with optimized multiplicative depth. Signals and operations are in $\mathbb{F}_{2^4}$, where $\otimes$ is multiplication, $\oplus$ is addition, $\bigcirc^2$ is squaring, $\bigcirc^4$ swaps $\mathbb{F}_{2^2}$ components, $\otimes\_$ scales by the given constant, $\Theta(\cdot)$ computes Θ (cf. Figure 2a), and $\odot$ is a pointwise multiplication of $\mathbb{F}_{2^2}$ components (cf. Figure 2b).

$\Gamma^4 = \gamma_0 Z^4 + \gamma_1 Z$ and $\Phi = \phi_1 Z^4 + \phi_0 Z$. After lifting the $\mathbb{F}_{2^2}$ element θ into $\mathbb{F}_{2^4}$ as

$$\Theta = \theta\pi^{-1}\pi = \theta\pi^{-1}\left(Z^4 + Z\right) = \theta\pi^{-1}Z^4 + \theta\pi^{-1}Z, \tag{10}$$

we rewrite equation (9) as

$$\Gamma^{-1}\Phi = \Theta\left(\Gamma^4\Phi\right), \tag{11}$$

which gives us the optimal multiplicative depth of two because

$$\operatorname{dep}\left(\Theta\left(\Gamma^4\Phi\right)\right) = \max\left(\operatorname{dep}(\Theta), \operatorname{dep}\left(\Gamma^4\Phi\right)\right) + 1 = \max(1,1) + 1 = 2. \tag{12}$$

Please note here that the multiplication with Θ is done in a pointwise manner and only requires two $\mathbb{F}_{2^2}$ multiplications between θ and the respective $\mathbb{F}_{2^2}$ components of $\Gamma^4\Delta$, making it cheaper than a normal $\mathbb{F}_{2^4}$ multiplication which requires three $\mathbb{F}_{2^2}$ multiplications as seen in equation (4).

Figure 3 shows the structure of the $\mathbb{F}_{2^8}$ inversion optimized for multiplicative depth. The parts of the computation that were to the left of the aforementioned cut in the computation graph are unchanged from the original design shown in Figure 1. Most notably, the $\mathbb{F}_{2^4}$ inversion and two multiplications that were to the right of the cut have been completely replaced using equation (11). Now the design first raises the middle input to the 4th power by swapping the $\mathbb{F}_{2^2}$ components and multiplies the result with Γ_0 and Γ_1. Furthermore, the design computes Θ with the sub-circuit shown in Figure 2a, with a simplified expression

$$\theta\pi^{-1} = \left(\gamma_1\gamma_0\pi^3 + (\gamma_1+\gamma_0)^2\sigma\pi\right)^{-1} = \left(\gamma_1\gamma_0 + (\gamma_1+\gamma_0)^2\sigma\pi\right)^2. \tag{13}$$

Finally, the new design performs a multiplication of Θ with the results of the prior two multiplications. Due to the two $\mathbb{F}_{2^2}$ components of Θ being equivalent, the $\mathbb{F}_{2^4}$ multiplication simplifies to a pointwise multiplication as shown in Figure 2b.

A bit-level AES S-Box implementation using the new $\mathbb{F}_{2^8}$ inverter is given in Appendix B. It achieves a gate depth of 14, beating the previous record, while remaining small [BP12].

4 HPC3.1 – A Better HPC3 Gadget

In this section, we argue that limiting a PINI gadget to operate in $\mathbb{F}_2$ has significant implications on its area and randomness consumption in a masked implementation. With the goal of achieving a low-latency masked AES S-Box utilizing our improved inverter design, we investigate the $\mathbb{F}_2$-only HPC3 gadget and show how it can be adapted to work in arbitrary finite fields, while simultaneously improving its area requirements and giving it better pipelining properties.

4.1 Field-level versus Bit-level Masking

While on the surface the support for larger fields does not seem like a significant feature, it has a steep impact on the randomness requirements and gate counts. Consider the HPC1 gadget shown in Algorithm 1 and its application to the $\mathbb{F}_{2^2}$ multiplication from (6). One can either mask the $\mathbb{F}_{2^2}$ multiplication directly with HPC1 for the field $\mathbb{F}_{2^2}$, or first break down the $\mathbb{F}_{2^2}$ multiplication into $\mathbb{F}_2$ multiplications and additions as in (6), and then apply HPC1 gadgets and sharewise addition gadgets for $\mathbb{F}_2$.

Field-level Masking. The direct $\mathbb{F}_{2^2}$ masking of a $\mathbb{F}_{2^2}$ multiplication requires $2\left(d\left(d+1\right)/2+r\left(d\right)\right)$ random bits and 2 refresh gadgets as each bit of A_i can be refreshed individually. The $\left(d+1\right)^2$ multiplications in $\mathbb{F}_{2^2}$, *i.e.,* $A_i \cdot B'_j$ in Algorithm 1, need $3\left(d+1\right)^2$ AND gates and $2\left(d+1\right)+2\left(d+1\right)^2$ XOR gates. The $2d\left(d+1\right)$ additions in $\mathbb{F}_{2^2}$ require 2 XOR gates each for a total of $4d\left(d+1\right)$ XOR gates, and similarly, the $\left(d+1\right)^2$ register operations in $\mathbb{F}_{2^2}$ require a total of $2\left(d+1\right)^2$ registers. Therefore, the overall cost is $d\left(d+1\right)+2r\left(d\right)$ random bits, 2 refresh gadgets, $3\left(d+1\right)^2$ AND gates, $2\left(d+1\right)\left(3d+2\right)$ XOR gates and $2\left(d+1\right)^2$ registers.

Bit-level Masking. In contrast, bit-blasting the $\mathbb{F}_{2^2}$ multiplication and then masking the computation requires 4 sharewise $\mathbb{F}_2$ additions, *i.e.,* $4\left(d+1\right)$ XOR gates, and 3 masked $\mathbb{F}_2$ multiplications using the $\mathbb{F}_2$ version of HPC1. Each $\mathbb{F}_2$ HPC1 gadget needs $d\left(d+1\right)/2+r(d)$ random bits, a refresh gadget, $\left(d+1\right)^2$ AND gates, $2d\left(d+1\right)$ XOR gates and $\left(d+1\right)^2$ registers. This brings the total cost to $3d\left(d+1\right)/2+3r(d)$ random bits, 3 refresh gadgets, $3\left(d+1\right)^2$ AND gates, $2\left(d+1\right)\left(3d+2\right)$ XOR gates, and $3\left(d+1\right)^2$ registers.

When masking a $\mathbb{F}_{2^2}$ multiplication, the first approach is clearly superior. While matching the number of AND and XOR gates, it requires significantly less random bits, registers, and refresh gadgets. Moreover, it appears that this trend continues for operations in larger $\mathbb{F}_{2^n}$ fields. Due to the algebraic structure of the AES S-Box, a low-latency masked implementation should use a gadget that achieves single clock cycle latency like HPC3, while being applicable to larger fields like HPC1.

4.2 Design, Optimization, Correctness and Security of HPC3.1

In Algorithm 3, we present HPC3.1, an improved version of the original HPC3 gadget introduced by Knichel and Moradi [KM22]. We incorporate several improvements over its predecessor shown in Algorithm 2.

Support for Generic Fields $\mathbb{F}_q$. It turns out that the negation of A_i while computing $W_{i,j}$ in HPC3 is unnecessary, and can be removed without any negative consequences. Although this changes the value of $U_{i,j}$ in the original HPC3 from $R_{i,j} \oplus P_{i,j} \oplus A_i \wedge B_j$ to $P_{i,j} \oplus A_i \wedge B_j$, it does not have any impact on the d-PINI property in the glitch-extended probing model. Finally, since there are no more $\mathbb{F}_2$-specific operations in Algorithm 2, one can replace all AND gates with field multiplications and all XOR gates with field additions. This is enough for the gadget to work in any $\mathbb{F}_{2^n}$ field. To preserve correctness in any

Algorithm 3: Generic HPC3.1 multiplication gadget

```
   Input : Sharing $\langle A_0, \dots, A_d \rangle \in \mathbb{F}_q^{d+1}$ of $A \in \mathbb{F}_q$,
           sharing $\langle B_0, \dots, B_d \rangle \in \mathbb{F}_q^{d+1}$ of $B \in \mathbb{F}_q$,
           masks $\langle R_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$,
           masks $\langle P_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$
   Output: Sharing $\langle C_0, \dots, C_d \rangle \in \mathbb{F}_q^{d+1}$ of $C = A \cdot B$
1  for $i$ from 0 to $d$ do
2      for $j$ from $i+1$ to $d$ do
3          $R_{j,i} \leftarrow R_{i,j}, P_{j,i} \leftarrow -P_{i,j}$;          // Randomness aliases
4  for $i$ from 0 to $d$ do
5      $V_{i,i} \leftarrow B_i, W_{i,i} = 0$;                          // Fictive same-domain terms
6      for $j$ from 0 to $d$ with $j \neq i$ do
7          $V_{i,j} \leftarrow R_{i,j} + B_j$;                         // Blinded multiplicands
8          $W_{i,j} \leftarrow P_{i,j} - A_i \cdot R_{i,j}$;                 // Correction terms
9      $C_i = \mathrm{Reg}(A_i) \cdot \left(\sum_{j=0}^{d} \mathrm{Reg}(V_{i,j})\right) + \left(\sum_{j=0}^{d} \mathrm{Reg}(W_{i,j})\right)$;
10 return $\langle C_0, \dots, C_d \rangle$;
```

finite field $\mathbb{F}_{p^n}$, one must additionally enforce that $P_{j,i} = -P_{i,j}$ for $j > i$, since additive inverse is only an identity operation in $\mathbb{F}_{2^n}$.

Factoring out the A_i Term. For our second improvement, notice that the same A_i is multiplied with all different $V_{i,j}$ in Algorithm 2 when computing the terms $U_{i,j}$. This leads to the algorithm requiring $(d+1) + (d+1)(2d)$ multiplications. However, since all the $A_i \cdot V_{i,j}$ multiplications share the same multiplicand A_i, one can factor it out and instead first sum over all $V_{i,j}$ before multiplying with A_i. The multiplication with B_i can also be integrated into this sum by adding a fictitious $V_{i,i} = B_i$. Therefore, by computing $A_i \cdot \sum_{j=0} V_{i,j}$ instead of what was done in the original HPC3 gadget in Algorithm 2, the optimized generic gadget HPC3.1 shown in Algorithm 3 replaces $(d+1)^2$ multiplications with only $(d+1)$ multiplications, requiring only $(d+1)^2$ multiplications overall. Of course, the values of A_i and $V_{i,j}$ must still be placed in registers, so there are no direct savings in register counts, *i.e.,* HPC3.1 still requires $2(d+1)^2$ registers. However, since HPC3.1 puts both input sharings $\langle A_0, \dots, A_d \rangle$ and $\langle B_0, \dots, B_d \rangle$ into registers, it effectively pipelines them, making their signals available at an additional clock cycle latency. This pipelining effect of the HPC3.1 gadget has the potential of sharing these pipelining registers with other parts of the overall design, e.g., other HPC3.1 gadgets that share an input. As for additions, HPC3.1 requires the same number of field additions as the original HPC3 gadget, that is, $4d(d+1)$.

Next, we prove that the new HPC3.1 gadget is indeed correct and fulfills the d-PINI property in the glitch-extended probing model. The proof consists of two parts, where we first show that the HPC3.1 gadget computes a correct multiplication, and afterwards prove that it also fulfills the d-PINI property.

Theorem 1. *Let $\mathbf{A} = \langle A_0, \dots, A_d \rangle \in \mathbb{F}_q^{d+1}$ be a sharing of $A \in \mathbb{F}_q$, $\mathbf{B} = \langle B_0, \dots, B_d \rangle \in \mathbb{F}_q^{d+1}$ be a sharing of $B \in \mathbb{F}_q$. Furthermore, let $\mathbf{R} = \langle R_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$ and $\mathbf{P} = \langle P_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$ be tuples of uniform and independent random elements in $\mathbb{F}_q$. Finally, let the sharing $\mathbf{C} = \langle C_0, \dots C_d \rangle \in \mathbb{F}_q^{d+1}$ of $C \in \mathbb{F}_q$ and the intermediate values $\mathbf{T}$ be computed according to Algorithm 3. The masked computation $\psi_{HPC3.1} : \langle \mathbf{A} \cup \mathbf{B}, \mathbf{R} \cup \mathbf{P} \rangle \mapsto \langle \mathbf{C}, \mathbf{T} \rangle$ implements a correct $\mathbb{F}_q$ multiplication and is d-PINI in the glitch-extended probing model.*

Proof. (Correctness.) The generic HPC3.1 gadget computes $C_i = A_i \cdot \left(\sum_{j=0}^{d} V_{i,j}\right) + \left(\sum_{j=0}^{d} W_{i,j}\right)$, where $V_{i,i} = B_i$ and $W_{i,i} = 0$, and similarly $V_{i,j} = R_{i,j} + B_j$ and $W_{i,j} = P_{i,j} - A_i \cdot R_{i,j}$ for $i \neq j$. We summarize these two cases by introducing fictitious values $R_{i,i} = 0$ and $P_{i,i} = 0$. Expanding C_i, we get

$$\begin{aligned} C_i = A_i \cdot \left(\sum_{j=0}^{d} V_{i,j}\right) + \left(\sum_{j=0}^{d} W_{i,j}\right) &= \sum_{j=0}^{d} (A_i \cdot V_{i,j} + W_{i,j}) = \\ = \sum_{j=0}^{d} (A_i \cdot (R_{i,j} + B_j) + P_{i,j} - A_i \cdot R_{i,j}) &= \sum_{j=0}^{d} (A_i \cdot B_j + P_{i,j}) . \end{aligned}$$

Furthermore, expanding C_i in $C = \sum_{i=0}^{d} C_i$, we see that

$$\begin{aligned} C = \sum_{i=0}^{d} C_i = \sum_{i=0}^{d} \sum_{j=0}^{d} (A_i \cdot B_j + P_{i,j}) &= \sum_{i=0}^{d} \sum_{j=0}^{d} A_i \cdot B_j + \sum_{i=0}^{d} \sum_{j=0}^{d} P_{i,j} = \\ = \left(\sum_{i=0}^{d} A_i\right) \left(\sum_{j=0}^{d} B_j\right) + \sum_{i=0}^{d-1} \sum_{j=i+1}^{d} P_{i,j} &+ \sum_{i=0}^{d-1} \sum_{j=i+1}^{d} P_{j,i} + \sum_{i=0}^{d} P_{i,i} = A \cdot B. \end{aligned}$$

(PINI.) In addition to the output sharing $\mathbf{C}$, a probing attacker also has access to all intermediate values $\mathbf{T}$, which includes all input values in $\mathbf{A}$, $\mathbf{B}$, $\mathbf{R}$ and $\mathbf{P}$, extended probes on explicit intermediates $\{V_{i,j}, W_{i,j} \mid i, j \in [0, d]\}$, as well as all implicit intermediates like individual additions and multiplications without a named result shown in Algorithm 3. In the glitch-extended probing model, some of those probes are contained (subsumed) within other more powerful probes, e.g., $\rho(A_i \cdot R_{i,j}) = \{A_i, R_{i,j}\} \subset \{P_{i,j}, A_i, R_{i,j}\} = \rho(W_{i,j})$. It is sufficient to analyze the set of all unsubsumed extended probes to prove that the masked computation $\psi_{HPC3.1}$ is d-PINI. Unsubsumed extended internal probes are $\rho(V_{i,j}) = \{R_{i,j}, B_j\}$ and $\rho(W_{i,j}) = \{P_{i,j}, A_i, R_{i,j}\}$ for $i \neq j$. The extended output probes $\rho(C_i)$ subsume everything in their computational cone, with $\rho(C_i) = \{A_i\} \cup \bigcup_{j=0}^{d} \{V_{i,j}, W_{i,j}\}$.

We first show that any set of output probes $\mathbf{Q}_{\text{out}} = \{\rho(C_i) \mid i \in I\}$ is perfectly simulated by $\mathbf{A}_I \cup \mathbf{B}_I$ under $\mathbf{A} \cup \mathbf{B}$. We give a constructive proof. Perfectly simulating $\rho(C_i)$ requires at least the domain i because $A_i, B_i \in \rho(C_i)$. When $\rho(C_j) \notin \mathbf{Q}_{\text{out}}$, additionally simulating $V_{i,j}, W_{i,j} \in \rho(C_i)$ does not require an additional domain, as the distribution of $V_{i,j}$ and $W_{i,j}$ is uniformly random because of the masks $R_{i,j}$ and $P_{i,j}$, which are not present elsewhere. If $\rho(C_j) \in \mathbf{Q}_{\text{out}}$, perfectly simulating $V_{i,j}$ and $V_{j,i}$ requires both B_j and B_i respectively. Similarly, perfectly simulating $W_{i,j}$ and $W_{j,i}$ requires A_i and A_j. However, since $\rho(C_j) \in \mathbf{Q}_{\text{out}}$ then also $j \in I$ by definition, and thus $B_j, A_j \in \mathbf{A}_I \cup \mathbf{B}_I$.

Finally, we show that any sets of output probes $\mathbf{Q}_{\text{out}}$ and internal probes $\mathbf{Q}_{\text{int}}$ can be perfectly jointly simulated by $\mathbf{A}_{I'} \cup \mathbf{B}_{I'}$ under $\mathbf{A} \cup \mathbf{B}$, where $I' \supseteq I$, $|I'| \leq |I| + |\mathbf{Q}_{\text{int}}|$ and I is defined as before. We prove this inductively, where, assuming that input shares $\mathbf{A}_{I'} \cup \mathbf{B}_{I'}$ simulate $\mathbf{Q}_{\text{out}} \cup \mathbf{Q}_{\text{int}}$ under $\mathbf{A} \cup \mathbf{B}$, we show that there is a set of indices I'' such that $\mathbf{A}_{I''} \cup \mathbf{B}_{I''}$ simulates $\mathbf{Q}_{\text{out}} \cup \mathbf{Q}_{\text{int}} \cup \{\rho(Q)\}$, with $Q \in \{V_{i,j}, W_{i,j} \mid i, j \in [0, d], i \neq j\}$ and $I'' \supseteq I, |I''| \leq |I'| + 1$. Here, if $\rho(Q) \in \mathbf{Q}_{\text{int}}$, then $I'' = I'$. Otherwise, we perform a case distinction based on Q:

- (Case $Q = V_{i,j}$, *i.e.*, $\rho(Q) = \{R_{i,j}, B_j\}$). If $\rho(C_j) \in \mathbf{Q}_{\text{out}}$, then $R_{i,j}$, $B_j \in \rho(Q)$ and $(R_{j,i} + B_i) \in \rho(C_j)$ must all be simulated jointly. Since $j \in I$ because $\rho(C_j) \in \mathbf{Q}_{\text{out}}$, the simulator can additionally require input share i for the simulation of $R_{j,i} + B_i$ and $R_{i,j} = R_{j,i}$, simulating the additional probe $\rho(Q)$. Therefore set $I'' = I' \cup \{i\}$. Otherwise, use the additional input share j for access to $B_j \in \rho(Q)$, *i.e.*, set $I'' = I' \cup \{j\}$. Either way, input shares $\mathbf{A}_{I''} \cup \mathbf{B}_{I''}$ perfectly simulate $\mathbf{Q}_{\text{out}} \cup \mathbf{Q}_{\text{int}} \cup \{\rho(Q)\}$ under $\mathbf{A} \cup \mathbf{B}$ and $|I''| \leq |I'| + 1$.

- (Case $Q = W_{i,j}$, *i.e.*, $\rho(Q) = \{P_{i,j}, A_i, R_{i,j}\}$). If $\rho(C_i) \in \mathbf{Q}_{\text{out}}$, then $P_{i,j}, A_i, R_{i,j} \in \rho(Q)$ and $(R_{i,j} + B_j) \in \rho(C_i)$ must all be simulated jointly. Here, $i \in I$ because $\rho(C_i) \in \mathbf{Q}_{\text{out}}$. Therefore, the simulator can additionally require input share j necessary for the joint simulation of $R_{i,j}$ and $R_{i,j} + B_j$, so set $I'' = I' \cup \{j\}$. Otherwise, set $I'' = I' \cup \{i\}$ to guarantee the simulator has access to A_i. In both cases, input shares $\mathbf{A}_{I''} \cup \mathbf{B}_{I''}$ perfectly simulate $\mathbf{Q}_{\text{out}} \cup \mathbf{Q}_{\text{int}} \cup \{\rho(Q)\}$ under $\mathbf{A} \cup \mathbf{B}$ and $|I''| \leq |I'| + 1$.

The HPC3.1 gadget with $d + 1$ shares is d-PINI in the glitch-extended probing model. □

5 Reusing Randomness in HPC1 and HPC3.1 Gadgets

Masking a circuit using HPC1 and HPC3.1 gadgets requires a lot of uniform random values per gadget execution. This has a direct impact on the area required for the finished circuit due to the need for fast pseudo-random number generators (PRNGs). Recently, Feldtkeller *et al.* [FKS+22] proposed a holistic method for reducing randomness requirements of masked S-Boxes implemented using DOM and HPC2 gadgets. Their method uses the structure of the given S-Box to argue when it is "safe" to reuse randomness, e.g., when gadget inputs are independent of each other. However, as they point out, their reuse strategy only preserves a restricted version of the respective SNI and PINI security notions.

In the following, we present an orthogonal randomness reuse method that preserves full d-PINI security without any downsides. The intuition behind this optimization is that both the HPC1 and HPC3.1 gadgets use part of their input masks to blind the sharing of B, and then perform any interactions with the sharing of A using the blinded values. Importantly, the randomness used for the blinding of an input sharing does not propagate to the output sharing of C, e.g., C_i does not functionally depend on the value of $R_{i,j}$ in HPC3.1 (cf. Theorem 1). This means that the randomness used for blinding is *local* to the gadget and could potentially be reused under certain circumstances.

Intuitively, a simple case for reusing randomness is when the same value B is multiplied with several different values A. Moreover, this exact situation appears multiple times in the design of both the original inverter design by Canright shown in Figure 1, as well as our modified design shown in Figure 3 In the following, we analyze this reuse case for both gadgets, and prove that the respective blinding values R_i and $R_{i,j}$ can be reused while preserving the d-PINI property in the glitch-extended probing model.

Theorem 2. *For $k \in [0, n)$ let $\langle A_{k,0}, \dots, A_{k,d}\rangle \in \mathbb{F}_q^{d+1}$ be sharings of $A_k \in \mathbb{F}_q$ and let $\langle P_{k,i,j} \mid 0 \leq i < j \leq d\rangle \in \mathbb{F}_q^{d(d+1)/2}$ be tuples of uniform and independent random elements in $\mathbb{F}_q$. Furthermore, let $\langle B_0, \dots, B_d\rangle \in \mathbb{F}_q^{d+1}$ be a sharing of $B \in \mathbb{F}_q$, and let $\langle R_0, \dots, R_d\rangle \in \mathbb{F}_q^{d+1}$ be a fresh, independent sharing of $0 \in \mathbb{F}_q$. The masked computation ψ_{nHPC1} : $\langle \mathbf{A}_{*,*} \cup \mathbf{B}, \mathbf{R} \cup \mathbf{P}_*\rangle \mapsto \langle \mathbf{C}_{*,*}, \mathbf{T}_*\rangle$ representing a sequence of HPC1 gadget applications*

$$\begin{aligned} \langle C_{k,0}, \dots, C_{k,d}\rangle = \mathit{HPC1}(&\langle A_{k,0}, \dots A_{k,d}\rangle, \langle B_0, \dots B_d\rangle, \\ &\langle R_0, \dots, R_d\rangle, \langle P_{k,i,j} \mid 0 \leq i < j \leq d\rangle) \end{aligned} \tag{14}$$

is d-PINI in the glitch-extended probing model. It correctly computes the output sharings $\langle C_{k,0}, \dots, C_{k,d}\rangle \in \mathbb{F}_q^{d+1}$ of $C_k \in \mathbb{F}_q$ for $k \in [0, n)$, where $C_k = A_k \cdot B$.

Proof. (Sketch.) The correctness follows from the correctness of HPC1. In the interest of brevity, we just give the reasoning behind the preservation of the d-PINI property when sharing the blinding randomness, rather than repeating the bulk of Cassiers' proof [CGLS21] in a different notation. The extended output share probe for share index i is $\bigcup_{k=0}^{n-1} \rho(C_{k,i}) = \bigcup_{k=0}^{n-1} \bigcup_{j=0}^{d} \{P_{k,i,j} + A_{k,i} \cdot (B_j + R_j)\}$, with fictive terms $P_{k,i,i} = 0$ for all k, i used for brevity. Since all $P_{k,i,j}$ are different independent and uniformly random field elements of

$\mathbb{F}_q$ for $j \neq i$, they effectively isolate all individual HPC1 gadgets from each other and make their output probes trivially jointly simulatable by $\{A_{k,i} \mid 0 \leq k < n\}$ under $\mathbf{A}_{*,*} \cup \mathbf{B}$. The joint simulation for output share index probes $\mathbf{Q}_{\text{out}}$ follows the same arguments as for a single HPC1 gadget.

For the additional simulation of internal probes only $\rho\left(B_i + R_i\right) = \{B_i, R_i\}$ and $\rho\left(V_{k,i,j}\right) = \{P_{k,i,j}, A_{k,i}, B_j + R_j\}$ must be considered, as everything else is subsumed by the more powerful output probes. For the probe $\rho\left(B_i + R_i\right)$, the simulator must have access to B_i and thus domain i. The simulation of probe $\rho\left(V_{k,i,j}\right)$ requires access to domain i since $A_{k,i} \in \rho\left(V_{k,i,j}\right)$. If $\rho\left(C_{k,j}\right) \in \mathbf{Q}_{\text{out}}$, then $-P_{k,i,j} + A_{k,j} \cdot (B_i + R_i) = V_{k,j,i} \in \rho\left(C_{k,j}\right)$ must be jointly simulated with $P_{k,i,j}, A_{k,i}, (B_j + R_j) \in \rho\left(V_{k,i,j}\right)$, which requires domains i and j but domain j is already available due to $\rho\left(C_{k,j}\right) \in \mathbf{Q}_{\text{out}}$. If $\rho\left(B_j + R_j\right) \in \mathbf{Q}_{\text{int}}$ then $(B_j + R_j) \in \rho\left(V_{k,i,j}\right)$ must be jointly simulated with $R_j \in \rho\left(B_j + R_j\right)$, which requires domain j but j is available since $\rho\left(B_j + R_j\right) \in \mathbf{Q}_{\text{int}}$. If $\rho\left(V_{k',i',j}\right) \in \mathbf{Q}_{\text{int}}$, then $A_{k,i}$, $A_{k',i'}$, and $B_j + R_j$ must be simulated jointly, however share index i' is available since $\rho\left(V_{k',i',j}\right) \in \mathbf{Q}_{\text{int}}$. This last combination is the reason the reuse of $\langle R_0, \ldots, R_d \rangle$ works when the input $\langle B_0, \ldots, B_d \rangle$ to all HPC1 gadgets is equivalent. Otherwise, for $d > 1$, one would need to jointly simulate $B_{k,j} + R_j$ and $B_{k',j} + R_j$, which requires simulating $(B_{k,j} + R_j) - (B_{k',j} + R_j) = B_{k,j} - B_{k',j}$. This is only possible when $B_{k,j} - B_{k',j}$ is a constant or with access to input domain j, in addition to i and i'. The latter would break the d-PINI guarantees. □

Theorem 3. *Let $\langle A_{k,0}, \ldots, A_{k,d} \rangle \in \mathbb{F}_q^{d+1}$ be valid sharings of $A_k \in \mathbb{F}_q$ for $k \in [0, n)$, and $\langle B_0, \ldots, B_d \rangle \in \mathbb{F}_q^{d+1}$ be a valid sharing of $B \in \mathbb{F}_q$. Furthermore, let $\langle R_{i,j} \mid 0 \leq i < j \leq d \rangle \in \mathbb{F}_q^{d(d+1)/2}$ and $\langle P_{k,i,j} \mid 0 \leq i < j \leq d \rangle \in \mathbb{F}_q^{d(d+1)/2}$, for $k \in [0, n)$, be $n+1$ tuples of uniform and independent random elements in $\mathbb{F}_q$. The masked computation $\psi_{nHPC3.1}$: $\langle \mathbf{A}_{*,*} \cup \mathbf{B}, \mathbf{R} \cup \mathbf{P}_* \rangle \mapsto \langle \mathbf{C}_{*,*}, \mathbf{T}_* \rangle$ representing a sequence of HPC3.1 gadget applications*

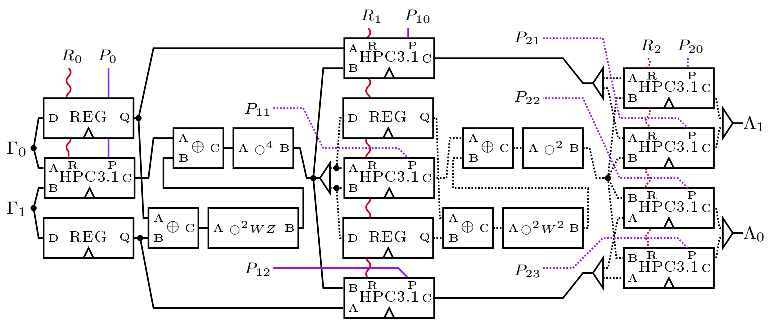
$$\begin{aligned}\langle C_{k,0}, \ldots, C_{k,d} \rangle = \mathit{HPC3.1}\left(\langle A_{k,0}, \ldots A_{k,d} \rangle, \langle B_0, \ldots B_d \rangle,\right.\\ \left.\langle R_{i,j} \mid 0 \leq i < j \leq d \rangle, \langle P_{k,i,j} \mid 0 \leq i < j \leq d \rangle\right)\end{aligned} \tag{15}$$

is d-PINI in the glitch-extended probing model. It correctly computes the output sharings $\langle C_{k,0}, \ldots, C_{k,d} \rangle \in \mathbb{F}_q^{d+1}$ of $C_k \in \mathbb{F}_q$ for $k \in [0, n)$, where $C_k = A_k \cdot B$.

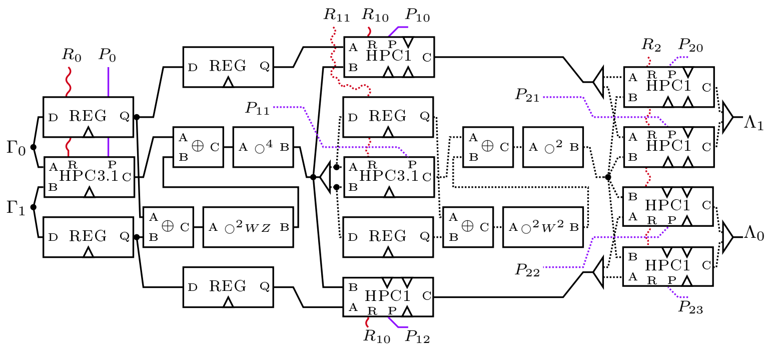
Proof. (Sketch.) The correctness follows from the correctness of HPC3.1. The proof of the d-PINI property follows the same structure as the proof for HPC3.1 itself. The argument for the simulation of outputs hinges on elements of $\rho\left(C_{k,i}\right)$ and $\rho\left(C_{k',i}\right)$ being isolated through different randomness $P_{k,i,j}$ and $P_{k',i,j}$, belonging to domain i, e.g., $\{A_{k,i}, A_{k',i}, B_i\}$, or belonging to both outputs and thus not granting any additional information, e.g., $(B_j + R_{i,j}) \in \rho\left(C_{k,i}\right) \cap \rho\left(C_{k',i}\right)$. If input $\langle B_0, \ldots, B_d \rangle$ were not the seme across all HPC3.1 instances and each got a different input $\langle B_{k,0}, \ldots, B_{k,d} \rangle$ for $k \in [0, n)$, then $B_{k,j} + R_{i,j}$ and $B_{k',j} + R_{i,j}$ would need to be simulated jointly, like in the proof of Theorem 2. This requires either $B_{k,j} - B_{k',j}$ to be a constant or the additional input share index $j \neq i$ for the simulation of $\rho\left(C_{k,i}\right) \cup \rho\left(C_{k',i}\right)$ under $\mathbf{A}_{*,*} \cup \mathbf{B}$, which would violate the d-PINI proprety. As for the internal probes, the arguments remain the same as in the proof of Theorem 1, with minor changes to replace A_i, $P_{i,j}$, and $W_{i,j}$ with $A_{k,i}$, $P_{k,i,j}$, and $W_{k,i,j}$ respectively. □

6 Masking the $\mathbb{F}_{2^8}$ Inversion

With the d-PINI gadgets HPC1 and HPC3.1 in hand, we are ready to mask the new design of the $\mathbb{F}_{2^8}$ inversion shown in Figure 3. Because we have presented the first non-trivial inverter design with $\operatorname{dep}\left(G^{-1}\right) = 3$, we are primarily interested in masked implementations achieving a latency of only three clock cycles. However, in order to gauge the quality



(a) All-HPC3.1 masked $\mathbb{F}_{2^8}$ inverter with a latency of *three* clock cycles



(b) HPC1 and HPC3.1 masked $\mathbb{F}_{2^8}$ inverter with a latency of *four* clock cycles

Figure 4: Masked implementation of the new $\mathbb{F}_{2^8}$ inverter using HPC1 gadgets, HPC3.1 gadgets, pipelining registers and trivially masked linear operations. Solid lines represent tuples of $\mathbb{F}_{2^4}$ elements while dotted lines show tuples of $\mathbb{F}_{2^2}$ elements.

of the inverter itself and have a better comparison against related work, we additionally investigate designs that achieve a latency of four clock cycles using both the old (cf. Figure 1) and new (cf. Figure 3) $\mathbb{F}_{2^8}$ inverter design.

The HPC1 and HPC3.1 gadgets have different cost characteristics, with HPC3.1 requiring more area and randomness in general, while HPC1 introduces additional output latency when both inputs arrive at the same time. Complicating matters further, there are many possible ways of reusing blinding randomness that must be taken into account. In the following we elaborate the design constraints, forced design features and optimization oportunities, resulting in three candidate designs.

6.1 Forced Design Features for a Latency of Three Clock Cyles

Aiming for the minimal latency of three clock cycles requires the use of the new $\mathbb{F}_{2^8}$ inverter and constrains the design space significantly. For example, the very first $\mathbb{F}_{2^4}$ multiplication between Γ_0 and Γ_1 in Figure 3 cannot use a HPC1 gadget and achieve the optimal latency, as the output would already have a latency of two clock cycles. Since the rest of the circuit has two multiplication stages needing at least one clock cycle each, the overall latency would exceed three clock cycles. Therefore, any latency-optimal design based on HPC1 and HPC3.1 gadgets must use HPC3.1 for this $\mathbb{F}_{2^4}$ multiplication.

For similar reasons, the $\mathbb{F}_{2^2}$ multiplication between γ_0 and γ_1 in Figure 2a when computing Θ, must also use a HPC3.1 gadget, since its inputs have a latency of one, and its outputs must achieve a latency of two to stay under three clock cycles overall.

Finally, the two pointwise multiplications computing Λ_0 and Λ_1 in Figure 3 must be implemented using HPC3.1 $\mathbb{F}_{2^2}$ multiplication gadgets, as neither of their inputs can have a latency of one clock cycle due to multiplicative depth. Moreover, these HPC3.1 gadgets force a randomness reuse pattern for optimal designs because they share the multiplicand θ, cf. equation (9) and Figure 2. Therefore, according to Theorem 3, all these HPC3.1 gadgets can share the randomness $\langle R_{i,j} \mid 0 \leq i < j \leq d \rangle$ used to blind the sharing of θ (the second input sharing), while preserving the d-PINI property in the glitch-extended probing model. This randomness reuse effectively reduces the randomness cost of the masked inverter by $3d\,(d+1)$ bits.

6.2 An All-HPC3.1 Design

The first masked $\mathbb{F}_{2^8}$ inverter design we present uses HPC3.1 gadgets for the two $\mathbb{F}_{2^4}$ multiplications between the output of $\bigcirc^4$ and Γ_0, respectively Γ_1. While HPC3.1 gadgets do have a larger area and randomness than HPC1 gadgets, they make up for it with good randomness reuse possibilities inside the $\mathbb{F}_{2^8}$ inversion shown in Figure 3. Since both HPC3.1 gadgets share an input, we can directly reuse the randomness necessary to blind the output of $\bigcirc^4$, cutting down the randomness cost by $2d\,(d+1)$ bits. Moreover, if we look at the structure of the Θ computation, we see that the upper and lower $\mathbb{F}_{2^2}$ parts of the $\bigcirc^4$ output are multiplied, as seen in Figure 2a. This means that we can reuse the randomness used to blind, e.g., the lower $\mathbb{F}_{2^2}$ portion in the the two aforementioned $\mathbb{F}_{2^4}$ multiplications, saving an additional $d\,(d+1)$ random bits. Overall the design uses $16d\,(d+1)$ random bits, 3 HPC3.1 $\mathbb{F}_{2^4}$ multiplications and 5 HPC3.1 $\mathbb{F}_{2^2}$ multiplications.

Figure 4a shows the structure of the masked all-HPC3.1 $\mathbb{F}_{2^8}$ inverter. Due to the pipelining properties of HPC3.1 gadgets, all shown pipelining registers are shared with a HPC3.1 gadget and thus *free*. Red wavy lines indicate blinding randomness and show its reuse across gadgets, while purple lines show the non-local randomness consumed by HPC3.1 gadgets.

6.3 Masking the new $\mathbb{F}_{2^8}$ Inverter with a Latency of Four Clock Cycles

Moving away from the minimal latency of three clock cycles achievable through HPC1 and HPC3.1 gadgets, we observe that there are no more forced design features and we are free to replace any HPC3.1 gadget from the previous design with a hopefully cheaper HPC1 implementation. However, some replacements are objectively better than others. Through manual and machine-assisted analysis, we have identified the design shown in Figure 4b as the most promising candidate achieving a latency of four clock cycles while saving area and randomness at higher masking orders. For instance, one can use HPC1 gadgets for the $\mathbb{F}_{2^4}$ multiplication between the output of $\bigcirc^4$ and Γ_0, respectively Γ_1, all while being able to also reuse the blinding randomness by choosing the $\bigcirc^4$ output as the second input sharing, and adding pipelining registers for Γ_0 and Γ_1 so they arrive at the HPC1 gadgets after two clock cycles. This change creates an input latency imbalance in the four $\mathbb{F}_{2^2}$ multipliers required for the two pointwise multiplications in the new inverter design. In turn, this imbalance allows the use of HPC1 gadgets for these $\mathbb{F}_{2^2}$ multiplications, while saving area and reusing randomness like in the all-HPC3.1 implementation. Overall the design uses $6r\,(d) + 14d\,(d+1)$ random bits, one HPC3.1 multiplier for $\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^4}$, two $\mathbb{F}_{2^4}$ HPC1 multipliers and four $\mathbb{F}_{2^2}$ HPC1 multipliers. Compared to the all-HPC3.1 design, this design should need less area and save randomness starting at $d = 4$.

6.4 Better Masking of Canright's original $\mathbb{F}_{2^8}$ Inverter

Canright's inversion shown in Figure 1 is the basis for most efficient masked $\mathbb{F}_{2^8}$ inverters implemented in related work. There is also a body of work on rewriting the bit-level description of Canright's inverter to either get smaller unmasked circuits or flatter circuits for low intra-clock cycle latencies in unmasked designs. Boyar and Peralta [BP12] present

Algorithm 4: A bit-level implementation of $\mathbb{F}_{2^4}$ inversion

```
Input  : $\Gamma = g_3Z^4W^2 + g_2Z^4W + g_1ZW^2 + g_0ZW$
Output : $\Gamma^{-1} = \Lambda = l_3Z^4W^2 + l_2Z^4W + l_1ZW^2 + l_0ZW$
1  $a_0 \leftarrow g_1 + g_0$;
2  $a_1 \leftarrow g_3 + g_2$;
3  $b_0 \leftarrow g_2 \cdot g_0$;
4  $b_1 \leftarrow g_3 \cdot g_1$;
5  $c_0 \leftarrow a_0 + b_0$;
6  $c_1 \leftarrow a_1 + b_0$;
7  $d_0 \leftarrow g_0 + b_1$;
8  $d_1 \leftarrow g_2 + b_1$;
9  $e_0 \leftarrow g_3 \cdot c_0$;
10 $e_1 \leftarrow g_1 \cdot c_1$;
11 $f_0 \leftarrow a_1 \cdot d_0$;
12 $f_1 \leftarrow a_0 \cdot d_1$;
13 $l_3 \leftarrow a_0 + e_1$;
14 $l_2 \leftarrow g_0 + f_1$;
15 $l_1 \leftarrow a_1 + e_0$;
16 $l_0 \leftarrow g_2 + f_0$;
```

one such flat implementation, which implements the $\mathbb{F}_{2^4}$ inversion required in Canright's inverter with only 7 $\mathbb{F}_2$ multiplications and a multiplicative depth of $\text{dep}\left(\Gamma^{-1}\right) = 2$. Unfortunately, bit-level masking of their $\mathbb{F}_{2^4}$ inverter design leads to worse randomness use than a direct masking of Canright's circuit using masked $\mathbb{F}_{2^2}$ operations. However, the idea of using a bit-level masking for the $\mathbb{F}_{2^4}$ inverter is interesting. A design using at most six $\mathbb{F}_2$ multiplications with $\text{dep}\left(\Gamma^{-1}\right) = 2$ could break even in randomness consumption, while using less operations overall. We have searched for such a bit-level circuit using a SAT-solver based approach inspired by Stoffelen's prior work [Sto16]. The result of our search is the first known bit-level $\mathbb{F}_{2^4}$ inverter to achieve the optimal number of six $\mathbb{F}_2$ multiplications for a multiplicative depth of two, as shown in Algorithm 4. [1]

If we consider the design space of masked inverters based on Canright's design that achieve a latency of four clock cycles, there are several forced design features. For similar reasons as elaborated in Section 6.1, the $\mathbb{F}_{2^4}$ multiplication between Γ_0 and Γ_1 from Figure 1 must be implemented using a HPC3.1 gadget. The same goes for the two bit-level multiplications resulting in b_0 and b_1 shown in Algorithm 4. All other multiplications are unconstrained. Moreover, the four bit-level multiplications resulting in e_0, e_1, f_0, and f_1 in Algorithm 4 may also be implemented using the HPC2 gadget without a significant overhead, compared to $\mathbb{F}_{2^2}$ or $\mathbb{F}_{2^4}$ multiplications. Later, we refer to these four multiplications as *middle* multiplications, whereas the two $\mathbb{F}_{2^4}$ multiplications at the back of Figure 1 are referred to as *back* multiplications.

Different choices for the unconstrained field multiplications lead to slightly different area and randomness overheads. As an example, the two $\mathbb{F}_{2^4}$ back multiplications can share the blinding randomness when implemented using HPC3.1, while they may use less randomness starting at $d = 4$ when implemented using HPC1. The two middle $\mathbb{F}_2$ multiplications should use the least randomness when implemented with HPC2, but may require more area than implementing them with HPC1.

7 Evaluation

In this section we first present the synthesis, testing and formal verification workflow used when developing and evaluating all gadgets and designs proposed in this paper. Afterwards, we present the evaluation results for HPC3.1 field multiplications in $\mathbb{F}_2$, $\mathbb{F}_{2^2}$ and $\mathbb{F}_{2^4}$ for practically relevant masking orders. Afterwards we evaluate masked AES S-Box designs based on different masked $\mathbb{F}_{2^8}$ inverters presented in Section 6. Finally, we compare our work against state-of-the-art AES S-Box implementations and demonstrate its efficiency. The masked hardware designs, as well as the testing and formal verification code are open source and available at

https://github.com/vedadux/Three-Stage-AES

[1]Boyar and Peralta [BP12] found circuits with only 5 multiplications with $\text{dep}\left(\Gamma^{-1}\right) = 3$ and 7 multiplications with $\text{dep}\left(\Gamma^{-1}\right) = 2$ through a different method.

7.1 Synthesis, Testing and d-PINI Verification

In Sections 4.1 and 4.2 we give exact counts for the field operations needed to implement the HPC1 and HPC3.1 gadgets. Similarly, we give detailed descriptions of inverter designs in Section 6 accompanied with block diagrams shown in Figure 4. However, while these are a good indicator of the real hardware cost in general, it is often more illuminating to see their cost in terms of *gate equivalents* (GE) when synthesized in a real cell library. Additionally, since masked computations usually require a lot of randomness, each random bit that must be generated at runtime using a PRNG induces additional area overhead. A recent paper proposes unrolled Trivium as a cost-effective PRNG [CMM+24], so we incorporate its per-bit average cost into our area considerations.

We have synthesized, functionally tested, and verified the d-PINI property of all gadgets and designs discussed in this paper. All designs were implemented in SystemVerilog. We have opted for the open source synthesis flow where the designs are first translated to Verilog using the SV2V utility [SH], and then elaborated, optimized and synthesized using Yosys [WGK13]. The circuits are technology-mapped using ABC [SG], with NanGate45 as the target cell library. In this cell library, the area overhead required by the Trivium PRNG is 39.4 GE per random bit.

The functionality of all designs has been thoroughly tested both at the RTL and the netlist level using the open source simulator Verilator [Sny]. Moreover, we have verified that all presented designs are d-PINI using the formal verification tools VerifMSI [MT23] and, to a minor extent, SILVER [KSM20]. Here, we first parse the netlists and encode them for VerifMSI using its Python API and run their d-PINI verification. If the verification is successful, we know that the masked computation is d-PINI. Otherwise, we get a list of all leaks present in the design. While this has found real implementation errors throughout the design process, it also produces false d-PINI violations on occasion. In such cases, we have analyzed the leakage reports with a modified version of SILVER, since it performs an exact count of the probability distributions involved. For the final designs, all issues raised by VerifMSI were verified to be spurious.

7.2 Comparison with Other d-PINI Multiplication Gadgets

Table 1 shows area and randomness cost of masked field multiplications implemented with HPC1 [CGLS21], HPC2 [CGLS21], HPC3 [KM22], HPC3.1 (this work), HPC3o [CGM+24] (non-Toffoli variant) and HPC4 [CSV24]. We see a clear reflection of the operation counts mentioned in Sections 4.1 and 4.2, where the HPC1 gadget consumes less area and randomness than the corresponding HPC3.1 gadget. Therefore, a design that can either use a HPC1 or HPC3.1 gadget for a masked field multiplication is better off using HPC1 in most cases. For $\mathbb{F}_2$ multiplications, using HPC2 instead of HPC3.1 is also an option when aiming for designs that use less randomness. Possible exceptions include cases where blinding randomness can be shared across multiple HPC3.1 instances but not HPC1 gadgets due to latency constraints or HPC2 gadgets where randomness sharing does not seem possible at all. Comparing HPC3.1 to HPC3, we see that it is always better, although only by a small margin in $\mathbb{F}_2$. In larger fields, were HPC3 to remove the negations as mentioned in Section 4.2, this difference would be even more pronounced due to multiplications in $\mathbb{F}_{2^n}$ dominating the area cost, compared to $\mathbb{F}_2$ where they are implemented using cheap AND and NAND gates.

Comparison of HPC3.1 and HPC3o. Concurrently to our work on this paper, Cassiers *et al.* published the HPC3o gadget as part of the Compress [CGM+24] automated masking suite. Both HPC3.1 and HPC3o remove the negation of A_i, enabling the gadgets to work in arbitrary fields, while being d-PINI property in the glitch-extended probing model. The main difference between HPC3.1 and HPC3o is how they modify their HPC3 blueprint. Unlike HPC3.1, which factors out $\mathrm{Reg}\,(A_i)$ to save $(d+1)^2 - (d+1)$ multiplications over

Table 1: Cost of a masked multiplications using various HPC gadgets in different fields.

Gadget	Field	d	Random bits	Area (GE)	
				standalone	w. PRNG
HPC1 + Shared Zero	$\mathbb{F}_2$	1	2	56.3	135.1
		2	5	127.0	324.0
		3	10	217.3	611.3
		4	15	325.0	916.0
	$\mathbb{F}_{2^2}$	1	4	139.3	296.9
		2	10	309.0	703.0
		3	20	528.0	1316.0
		4	30	791.7	1973.7
	$\mathbb{F}_{2^4}$	1	8	386.0	701.2
		2	20	822.7	1610.7
		3	40	1397.0	2973.0
		4	60	2072.0	4436.0
HPC2	$\mathbb{F}_2$	1	1	82.3	121.7
		2	3	209.0	327.2
		3	6	392.7	629.1
		4	10	633.3	1027.3
HPC3o + Reg ($\mathbf{A}$)	$\mathbb{F}_2$	1	2	50.0	128.8
		2	6	135.0	371.4
		3	12	265.3	738.1
		4	20	435.0	1223.0
	$\mathbb{F}_{2^2}$	1	4	135.7	293.3
		2	12	368.0	840.8
		3	24	715.3	1660.9
		4	40	1173.0	2749.0
	$\mathbb{F}_{2^4}$	1	8	379.3	694.5
		2	24	1048.0	1993.6
		3	48	2006.7	3897.9
		4	80	3285.7	6437.7
HPC3	$\mathbb{F}_2$	1	2	69.3	148.1
		2	6	168.0	404.4
		3	12	302.7	775.5
		4	20	484.7	1272.7
HPC3.1	$\mathbb{F}_2$	1	2	64.7	143.5
		2	6	159.0	395.4
		3	12	293.3	766.1
		4	20	466.7	1254.7
	$\mathbb{F}_{2^2}$	1	4	167.3	324.9
		2	12	390.0	862.8
		3	24	705.3	1650.9
		4	40	1116.7	2692.7
	$\mathbb{F}_{2^4}$	1	8	437.7	752.9
		2	24	1002.0	1947.6
		3	48	1769.3	3660.5
		4	80	2791.3	5943.3
HPC4	$\mathbb{F}_2$	1	5	93.3	290.3
		2	15	242.0	833.0
		3	30	458.7	1640.7
		4	50	742.0	2712.0
	$\mathbb{F}_{2^2}$	1	10	251.3	645.3
		2	30	637.0	1819.0
		3	60	1205.7	3569.7
		4	100	1947.7	5887.7
	$\mathbb{F}_{2^4}$	1	20	693.0	1481.0
		2	60	1756.3	4120.3
		3	120	3287.0	8015.0
		4	200	5332.3	13212.3

HPC3, HPC3o instead puts the $A_i \cdot B_i$ term together with one of the cross-domain terms $W_{i,j}$, saving $(d+1)$ multiplications and pipelining registers over HPC3, as well as $(d+1)$ additions when working in $\mathbb{F}_{2^n}$ due to the additive inverse being the identity operation. We see a reflection of this in Table 1, where HPC3.1 is more efficient in larger fields and at larger protection orders, due to the domination of multiplication costs. While the register savings in HPC3o are great, they do not really help when masking the new $\mathbb{F}_{2^8}$ inversion since the inputs are almost always pipelined outside of the multiplication gadgets as seen in Figure 4. Importantly, the rewrites used in HPC3.1 and HPC3o are orthogonal and can be applied jointly for an even better gadget, cf. Appendix A.

Comparison of HPC3.1 and HPC4. Concurrently to our development of HPC3.1, Cassiers *et al.* [CSV24] presented HPC4, a low-latency secure multiplication gadget that similarly diverges from HPC3. It fulfils a more strict security property called Output (O)-PINI, which ensures secure self-composition in the presence of transition leakage. What this means is that, unlike HPC3.1 and other mentioned gadgets, the output of a HPC4 gadget can be connected to its input in the next clock cycle without issues. However, this comes at the cost of significant area and randomness overheads as shown in Table 1. Interestingly, because HPC4 and HPC3 have a similar structure, one can trivially apply the factorization trick from HPC3.1 to improve the HPC4 gadget while maintaining the O-PINI property.

Masked AES S-Box implementations based on inverters from Section 6 can circumvent transition issues by either (i) waiting for one clock cycle between rounds in the case of serial implementations, or (ii) feeding all S-Box instances with the constant $0 \in \mathbb{F}_{2^8}^{d+1}$ after the first clock cycle of each round in fully parallel implementations. Alternatively, one could also replace the back HPC3.1 and HPC1 multiplications with HPC4 to preserve optimal latency and achieve composable security in the presence of glitches and transitions.

7.3 Comparison with State-of-the-Art d-PINI AES S-Boxes

In the following, we compare our new d-PINI AES S-Boxes and highlight their differences. Additionally, we compare the new masked S-Box designs to state-of-the-art masked d-PINI AES S-Box implementations from the literature. Table 2 shows the latency, area, and randomness cost of the various d-PINI AES S-Box implementations.

Our d-PINI AES S-Boxes. As the first masked AES S-Box using the new $\mathbb{F}_{2^8}$ inverter, the all-HPC3.1 design from Section 6.2 is also the only one in our comparison that achieves a latency of three clock cycles. Moreover, it has relatively modest area and randomness costs. Among the two masked AES S-Boxes achieving a latency of four clock cycles, the design from Section 6.4 based on Canright's $\mathbb{F}_{2^8}$ inversion with the new bit-level $\mathbb{F}_{2^4}$ inversion performs slightly better across the board than design from Section 6.3, which is based on the new $\mathbb{F}_{2^8}$ inverter. Thus, we select the d-PINI AES S-Boxes based on Sections 6.2 and 6.4 for detailed comparison to the state-of-the-art.

Other d-PINI AES S-Boxes. Because of the trivial composability guaranteed by the d-PINI property, many recent papers propose automated synthesis of masked designs from regular unmasked hardware implementations at arbitrary protection orders using a library of d-PINI gadgets [KMMS22, WFP+24, CGM+24]. Most of these methods work based on a netlist representation of a circuit and can, therefore, not take advantage of field-level masking in the particular case of the AES S-Box. As discussed in Section 4.1, the limitation to bit-level masking leaves the automated methods AGEMA [KMMS22] and AGMNC [WFP+24] at a massive disadvantage, making their results wholly uncompetitive as seen in Table 2. Notable exceptions among state-of-the-art d-PINI AES S-Boxes are those proposed by Knichel and Moradi [KM22], which are the first to use HPC3, the ones proposed by Momin *et al.* [MCS22], where manual optimization was performed after automated masking, and those produced by Compress [CGM+24], which also has the capability to mask in fields other than $\mathbb{F}_2$. Nevertheless, our two selected masked S-Boxes require between 22% and 44.2% less PRNG-adjusted area than the designs by Momin *et al.* or Knichel and Moradi. Compress' Canright-based masked AES S-Box is of comparable costs to our results, where the all-HPC3.1 design saves between 0.9% and 6.9% PRNG-adjusted area and the design from Section 6.4 requires between 8.2% and 12.5% less PRNG-adjusted area. Better inverter designs, better multiplication gadgets and clever randomness reuse are vital for performant masked AES S-Box implementations.

7.4 Comparison with non-PINI masked AES S-Boxes

Efficient masking of the AES S-Box is a well researched topic. In Table 3, we present an assortment of various state-of-the-art masked AES S-Box designs together with their latency, randomness and area characteristics. Threshold Implementation (TI) is the longest standing class of masking techniques which split the target functions in such a way that each share of the output is completely independent of at least one input share, making them trivially resistent in the presence of glitches. Compared with the works of Cnudde *et al.* [CRB+16], Ueno *et al.* [UHA17] and Bilgin *et al.* [BGN+15], our proposed AES S-Boxes have a lower latency and randomness cost, leading to PRNG-adjusted area savings between 10% and 26.3%, all while being trivially composable. Furthermore, our

Table 2: Area and randomness cost of our masked d-PINI AES S-Boxes compared to state-of-the-art d-PINI designs, annotated with percentages showing how much cost is saved by the All-HPC3.1 design (black) and the HPC1-middle-HPC1-back design (gray).

AES S-Box	Latency (cycles)	d	Random bits			Area (GE)					
						standalone			with PRNG		
New $\mathbb{F}_{2^8}$ Inverter All-HPC3.1 (Section 6.2)	3	1	32	– – –	9.4%	1875	– – –	3.7%	3136	– – –	6.0%
		2	96	– – –	0.0%	4176	– – –	7.1%	7958	– – –	3.7%
		3	192	– – –	0.0%	7687	– – –	14.6%	15252	– – –	7.4%
		4	320	– – –	6.2%	11465	– – –	13.7%	24073	– – –	9.8%
New $\mathbb{F}_{2^8}$ Inverter Mixed HPC1 and HPC3.1 (Section 6.3)	4	1	34	5.9%	14.7%	1905	1.5%	5.2%	3244	3.3%	9.1%
		2	96	0.0%	0.0%	4126	−1.2%	6.0%	7908	−0.6%	3.1%
		3	192	0.0%	0.0%	7215	−6.5%	9.0%	14780	−3.2%	4.4%
		4	310	−3.2%	3.2%	10899	−5.2%	9.2%	23113	−4.2%	6.1%
Canright HPC1-middle HPC1-back (Section 6.4)	4	1	29	−10.3%	– – –	1806	−3.8%	– – –	2948	−6.4%	– – –
		2	96	0.0%	– – –	3880	−7.6%	– – –	7662	−3.9%	– – –
		3	192	0.0%	– – –	6563	−17.1%	– – –	14127	−8.0%	– – –
		4	300	−6.7%	– – –	9893	−15.9%	– – –	21713	−10.9%	– – –
Compress [CGM+24] Canright (with fields)[1]	4	1	36	11.1%	19.4%	1950	3.8%	7.4%	3368	6.9%	12.5%
		2	96	0.0%	0.0%	4560	8.4%	14.9%	8342	4.6%	8.2%
		3	192	0.0%	0.0%	8060	4.6%	18.6%	15624	2.4%	9.6%
		4	300	−6.7%	0.0%	12480	8.1%	20.7%	24300	0.9%	10.6%
Low-Latency HPC [KM22] HPC3[3]	4	1	68	52.9%	57.4%	1849	−1.4%	2.3%	4528	30.7%	34.9%
		2	204	52.9%	52.9%	4855	14.0%	20.1%	12892	38.3%	40.6%
		3	408	52.9%	52.9%	9261	17.0%	29.1%	25336	39.8%	44.2%
AGMNC [WFP+24] AND-XOR1[3]	6	1	66	51.5%	56.1%	2895	35.2%	37.6%	5495	42.9%	46.3%
		2	165	41.8%	41.8%	5745	27.3%	32.5%	12246	35.0%	37.4%
		3	330	41.8%	41.8%	9243	16.8%	29.0%	22245	31.4%	36.5%
		4	495	35.4%	39.4%	13314	13.9%	25.7%	32817	26.6%	33.8%
AGMNC [WFP+24] AND-XOR2[3]	6	1	33	3.0%	12.1%	3967	52.7%	54.5%	5267	40.5%	44.0%
		2	99	3.0%	3.0%	9078	54.0%	57.3%	12978	38.7%	41.0%
		3	198	3.0%	3.0%	16239	52.7%	59.6%	24040	36.6%	41.2%
		4	330	3.0%	9.1%	25469	55.0%	61.2%	38471	37.4%	43.6%
Handcrafting [MCS22] HPC2[2]	6	1	34	5.9%	14.7%	3213	41.6%	43.8%	4552	31.1%	35.2%
		2	102	5.9%	5.9%	6705	37.7%	42.1%	10723	25.8%	28.5%
		3	204	5.9%	5.9%	11515	33.2%	43.0%	19552	22.0%	27.7%
AGEMA [KMMS22] HPC1 Optimized Pipelined[1]	8	1	68	52.9%	57.4%	4263	56.0%	57.6%	6942	54.8%	57.5%
		2	170	43.5%	43.5%	7839	46.7%	50.5%	14537	45.3%	47.3%
		3	340	43.5%	43.5%	12085	36.4%	45.7%	25481	40.1%	44.6%
		4	510	37.3%	41.2%	16919	32.2%	41.5%	37013	35.0%	41.3%
AGEMA [KMMS22] HPC2 Optimized Pipelined[1]	8	1	34	5.9%	14.7%	5339	64.9%	66.2%	6678	53.0%	55.8%
		2	102	5.9%	5.9%	11205	62.7%	65.4%	15223	47.7%	49.7%
		3	204	5.9%	5.9%	19217	60.0%	65.8%	27254	44.0%	48.2%
		4	340	5.9%	11.8%	29267	60.8%	66.2%	42663	43.6%	49.1%

[1] Synthesized with Yosys to the NanGate45 design kit.
[2] Synthesized with Cadence to the TSMC-N65 design kit.
[3] Synthesized with Synopsys to the NanGate45 design kit.

designs are neck-and-neck with the well-known DOM design by Gross *et al.* [GMK16]. Importantly, DOM does not provide any composition guarantees, and constructs like the DOM-dep gadget have been shown to be trivially insecure by Moos *et al.* [MMSS19].

Masking schemes using dual-rail pre-charge logic such as LMDPL [SBHM20] or

Table 3: Area and randomness cost of state-of-the-art masked AES S-Boxes, annotated with percentages showing how much cost is saved by the All-HPC3.1 design (black) and the HPC1-middle-HPC1-back design (gray), cf. Table 2.

Method	AES S-Box	Latency (cycles)	d	Random bits		Area (GE)			
						standalone		with PRNG	
TI	[CRB+16][S1]	6	1	54	40.7% 46.3%	1872	−0.2% 3.5%	3999	21.6% 26.3%
			2	162	40.7% 40.7%	3662	−14.0% −6.0%	10044	20.8% 23.7%
	[UHA17][S5]	6	1	64	50.0% 54.7%	1342	−39.8% −34.6%	3863	18.8% 23.7%
	[BGN+15][S2]	4	1	32	0.0% 9.4%	2224	15.7% 18.8%	3484	10.0% 15.4%
DOM	[GMK16][C2]	5	1	18	−77.8% −61.1%	2600	27.9% 30.5%	3309	5.2% 10.9%
			2	54	−77.8% −77.8%	5300	21.2% 26.8%	7427	−7.2% −3.2%
LMDPL	[SBHM20][S4]	1	1	36	11.1% 19.4%	3480	46.1% 48.1%	4898	36.0% 39.8%
SESYM	[NGPM22][C3] BP	1	1	34	5.9% 14.7%	3980	52.9% 54.6%	5319	41.0% 44.6%
			2	102	5.9% 5.9%	9340	55.3% 58.5%	13358	40.4% 42.6%
	[NGPM22][C3] C	1	1	18	−77.8% −61.1%	7590	75.3% 76.2%	8299	62.2% 64.5%
			2	51	−88.2% −88.2%	14780	71.7% 73.7%	16789	52.6% 54.4%

[C] Synthesized with Cadence [S] Synthesized with Synopsys
[1] NanGate45 design kit [3] UMC 65nm design kit [5] TSMC-N65 design kit
[2] UMC 180nm design kit [4] GF 28nm design kit

SESYM [NGPM22] offer very low latency overheads of only one clock cycle. While this does use more area than HPC-based designs like ours, it is nevertheless a viable and performant alternative. However, they do rely on stronger assumptions about the physical properties of circuits, some of which have recently been questioned by Müller *et al.* [MLM24].

8 Conclusion

This paper improves the trivially composable masking of the AES S-Box through innovations on several fronts, from better $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^4}$ inverter designs, an improved HPC3.1 gadget for masked multiplications in arbitrary finite fields, all the way to sound mask reuse reducing the randomness cost. The resulting designs show a clear improvement over state-of-the-art masked AES S-Box designs, be it in latency, randomness cost or area. However, we argue that our contributions are not limited to the masked AES S-Box only; and most are of independent interest for both masked and unmasked hardware designs in general. We have shown how algebraic rewriting can be used to reduce multiplicative depth, yielding lower latencies for masked hardware designs, and potentially lower intra-clock cycle delays of unmasked implementations. Moreover, the generality of the HPC3.1 gadget could lend itself for low-latency masking of other cryptographic primitives that work with finite-field representations, e.g., Dilithium [DKL+18] or Kyber [BDK+18]. Similarly, we think that the sound reuse of randomness presented in this paper represents a crucial step towards low-randomness masked implementations that are d-PINI in the glitch-extended probing model, and could positively affect the masking of most cryptographic primitives, especially in the domain of lightweight symmetric cryptography.

Acknowledgements

This work was supported by the *Austrian Research Promotion Agency (FFG)* through the BRIDGE project AWARE.

References

[ADN$^+$22] Amund Askeland, Siemen Dhooghe, Svetla Nikova, Vincent Rijmen, and Zhenda Zhang. Guarding the first order: The rise of AES maskings. In *Smart Card Research and Advanced Applications – CARDIS*, pages 103–122, 2022.

[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 *Conference on Computer and Communications Security – CCS*, 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 *Advances in Cryptology – EUROCRYPT*, pages 616–648, 2016.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems – CHES*, pages 16–29, 2004.

[BDK$^+$18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *European Symposium on Security and Privacy – EuroS&P*, pages 353–367, 2018.

[BGN$^+$15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-offs for threshold implementations illustrated on AES. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34:1188–1200, 2015.

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, pages 178–189, 2010.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In *Information Security Conference – SEC*, pages 287–298, 2012.

[Can05] David Canright. A very compact s-box for AES. In *Cryptographic Hardware and Embedded Systems – CHES*, pages 441–455, 2005.

[CBR$^+$15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES s-box. In *Smart Card Research and Advanced Applications – CARDIS*, pages 259–272, 2015.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70:1677–1690, 2021.

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

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology – CRYPTO*, pages 398–412, 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 Commun. Cryptol.*, 1:4, 2024.

[CRB+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *Cryptographic Hardware and Embedded Systems – CHES*, pages 194–212, 2016.

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

[CSV24] Gaëtan Cassiers, François-Xavier Standaert, and Corentin Verhamme. Low-latency masked gadgets robust against physical defaults with application to ascon. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 603–633, 2024.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 238–268, 2018.

[DR98] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *Smart Card Research and Advanced Applications – CARDIS*, pages 277–284, 1998.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 89–120, 2018.

[FKS+22] Jakob Feldtkeller, David Knichel, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Randomness optimization for gadget compositions in higher-order masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 188–227, 2022.

[GC17] Ashrujit Ghoshal and Thomas De Cnudde. Several masked implementations of the boyar-peralta AES s-box. In *Progress in Cryptology – INDOCRYPT*, pages 384–402, 2017.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 1–21, 2018.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Conference on Computer and Communications Security – CCS*, page 3, 2016.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In *Topics in Cryptology – CT-RSA*, pages 95–112, 2017.

[GSM+19] Hannes Groß, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. First-order masking with only two random bits. In *Conference on Computer and Communications Security – CCS*, pages 10–23, 2019.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology – CRYPTO*, pages 463–481, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO*, pages 388–397, 1999.

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

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 589–629, 2022.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *Advances in Cryptology – ASIACRYPT*, pages 787–816, 2020.

[MCS22] Charles Momin, Gaëtan Cassiers, and François-Xavier Standaert. Handcrafting: Improving automated masking in hardware with manual optimizations. In *Constructive Side-Channel Analysis and Secure Design – COSADE*, pages 257–275, 2022.

[MLM24] Nicolai Müller, Daniel Lammers, and Amir Moradi. A deep analysis of two glitch-free hardware masking schemes SESYM and LMDPL. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 76–98, 2024.

[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 Trans. Cryptogr. Hardw. Embed. Syst.*, pages 256–292, 2019.

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In *Advances in Cryptology – EUROCRYPT*, pages 69–88, 2011.

[MT23] Quentin L. Meunier and Abdul Rahman Taleb. Verifmsi: Practical verification of hardware and software masking schemes implementations. In *Proceedings of the 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, Italy, July 10-12, 2023*, pages 520–527, 2023.

[NGPM22] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. Riding the waves towards generic single-cycle masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 693–717, 2022.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptol.*, 24:292–321, 2011.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology – CRYPTO*, pages 764–783, 2015.

[SBHM20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-latency hardware masking with application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 300–326, 2020.

[SG] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification.

[SGMT18] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. An inside job: Remote power analysis attacks on fpgas. In *Design, Automation & Test in Europe – DATE*, pages 1111–1116, 2018.

[SH] Zachary Snow and Tom Hawkins. sv2v: Systemverilog to verilog.

[Sny] Wilson Snyder. Verilator: Open-source systemverilog simulator and lint system.

[Sto16] Ko Stoffelen. Optimizing s-box implementations for several criteria using SAT solvers. In *Fast Software Encryption – FSE*, pages 140–160, 2016.

[UHA17] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward more efficient dpa-resistant AES hardware architecture based on threshold implementation. In *Constructive Side-Channel Analysis and Secure Design – COSADE*, pages 50–64, 2017.

[WFP$^+$24] Lixuan Wu, Yanhong Fan, Bart Preneel, Weijia Wang, and Meiqin Wang. Automated generation of masked nonlinear components: - from lookup tables to private circuits. In *Applied Cryptography and Network Security – ACNS*, pages 319–339, 2024.

[WGK13] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

Algorithm 5: Generic HPC3.2 multiplication gadget

```
   Input  : Sharing $\langle A_0, \ldots, A_d \rangle \in \mathbb{F}_q^{d+1}$ of $A \in \mathbb{F}_q$,
            sharing $\langle B_0, \ldots, B_d \rangle \in \mathbb{F}_q^{d+1}$ of $B \in \mathbb{F}_q$,
            masks $\langle R_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$,
            masks $\langle P_{i,j} \mid 0 \le i < j \le d \rangle \in \mathbb{F}_q^{d(d+1)/2}$
   Output : Sharing $\langle C_0, \ldots, C_d \rangle \in \mathbb{F}_q^{d+1}$ of $C = A \cdot B$
1  for $i$ from 0 to $d$ do
2      for $j$ from $i+1$ to $d$ do
3          $R_{j,i} \leftarrow R_{i,j}, P_{j,i} \leftarrow -P_{i,j}$;          // Randomness aliases
4  for $i$ from 0 to $d$ do
5      for $j$ from 0 to $d$ with $j \neq i$ do
6          $V_{i,j} \leftarrow R_{i,j} + B_j$;                                   // Blinded multiplicands
7          if $j \equiv i+1 \mod d+1$ then
8              $W_{i,j} \leftarrow P_{i,j} - A_i \cdot (R_{i,j} - B_i)$ ;          // Special correction terms
9          else
10             $W_{i,j} \leftarrow P_{i,j} - A_i \cdot R_{i,j}$;                  // Normal correction terms
11     $C_i = \mathrm{Reg}(A_i) \cdot \left(\sum_{j \in [0,d] \setminus \{i\}} \mathrm{Reg}(V_{i,j})\right) + \left(\sum_{j \in [0,d] \setminus \{i\}} \mathrm{Reg}(W_{i,j})\right)$;
12 return $\langle C_0, \ldots, C_d \rangle$;
```

A HPC3.2 – An Even Better Multiplication Gadget

As mentioned in Section 7, one can adapt the rewrite trick from HPC3o to HPC3.1, which we missed during its initial design process and subsequent d-PINI proof. Algorithm 5 shows a gadget called HPC3.2 that incorporates a HPC3o-style cancellation term rewrite into HPC3.1. Let $i' = i + 1 \mod d + 1$. In HPC3.2 the $A_i \cdot B_i$ term becomes a summand of C_i through $W_{i,i'}$ because

$$W_{i,i'} = P_{i,i'} - A_i \cdot (R_{i,i'} - B_i) = P_{i,i'} - A_i \cdot R_{i,i'} + A_i \cdot B_i, \tag{16}$$

compared to HPC3.1, where it becomes a summand of C_i through $A_i \cdot \sum_{j=0}^{d} V_{i,j}$ with $V_{i,i} = B_i$. All other parts of HPC3.2 stay the same as in HPC3.1.

In addition to saving $(d + 1)$ field element registers, this change also saves $(d + 1)$ field additions for $\mathbb{F}_{2^n}$ where field element negation is the identity operation due to $R_{i,j} - B_i = R_{i,j} + B_i$ which is shared with $V_{i,j}$, and thus saved.

The d-PINI proof for HPC3.2 is the same as in Theorem 1, except that $W_{i',j}$ must be handled more carefully, but otherwise behaves the same as any other $W_{i,j}$. Moreover, randomness reuse across HPC3.2 gadgets works the same as for HPC3.1 gadgets.

B Bit-level AES S-Box Using the New $\mathbb{F}_{2^8}$ Inverter

In the following, we give a bit-level implementation of the AES S-Box which breaks Boyar and Peralta's [BP10] record for the shallowest AES S-Box implementation. Their implementation has a gate depth of 16, while our design has a gate depth of 14, while only mildly increasing the circuit size. Our construction is based on the new $\mathbb{F}_{2^8}$ inverter shown in Figure 3, where all operations are broken down into their bit-level variants. This leaves us with the structure depicted in Figure 5.

In the following, we apply an optimization method inspired by Stoffelen [Sto16] to the linear layers of the three-stage S-Box, and report the results. Using this SAT-based method, we found an S-Box design with 132 gates and gate depth 19, another circuit with 134 gates and gate depth 15, as well as a circuit with 139 gates and gate depth 14. This

Algorithm 6: A bit-level implementation of AES S-Box with gate depth of 14

```
    Input  : $\langle x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 \rangle$
    Output : $\langle y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7 \rangle$
    // $L_0$ layer
1   $p_0 \leftarrow x_1 \oplus x_3$;
2   $p_1 \leftarrow x_5 \oplus x_6$;
3   $b_4 \leftarrow x_0 \oplus p_1$;
4   $p_2 \leftarrow x_2 \oplus x_5$;
5   $a_7 \leftarrow x_2 \oplus x_7$;
6   $a_0 \leftarrow x_4 \oplus b_4$;
7   $a_5 \leftarrow x_1 \oplus x_7$;
8   $a_6 \leftarrow x_4 \oplus x_7$;
9   $a_8 \leftarrow x_2 \oplus x_4$;
10  $b_2 \leftarrow p_0 \oplus a_6$;
11  $b_5 \leftarrow p_0 \oplus p_2$;
12  $c_0 \leftarrow x_7 \oplus b_5$;
13  $p_3 \leftarrow x_2 \oplus a_5$;
14  $b_7 \leftarrow p_1 \oplus b_2$;
15  $a_4 \leftarrow x_1 \oplus b_4$;
16  $b_8 \leftarrow p_2 \oplus a_6$;
17  $a_3 \leftarrow x_7 \oplus b_4$;
18  $c_1 \leftarrow x_1 \oplus c_0$;
19  $b_6 \leftarrow p_1 \oplus b_5$;
20  $a_2 \leftarrow x_4 \oplus p_3$;
21  $b_1 \leftarrow x_0 \oplus b_2$;
22  $b_3 \leftarrow b_4 \oplus b_5$;
23  $c_2 \leftarrow a_7 \oplus b_7$;
24  $c_3 \leftarrow a_6 \oplus b_6$;
25  $a_1 \leftarrow b_4 \oplus p_3$;
    // $N_0$ layer
26  $d_0 \leftarrow a_0 \wedge x_0$;
27  $d_1 \leftarrow a_1 \wedge b_1$;
28  $d_2 \leftarrow a_2 \wedge b_2$;
29  $d_3 \leftarrow a_3 \wedge b_3$;
30  $d_4 \leftarrow a_4 \wedge b_4$;
31  $d_5 \leftarrow a_5 \wedge b_5$;
32  $d_6 \leftarrow a_6 \wedge b_6$;
33  $d_7 \leftarrow a_7 \wedge b_7$;
34  $d_8 \leftarrow a_8 \wedge b_8$;
    // $L_1$ layer
35  $q_0 \leftarrow d_6 \oplus d_2$;
36  $q_1 \leftarrow d_7 \oplus d_3$;
37  $q_2 \leftarrow q_0 \oplus q_1$;
38  $q_3 \leftarrow d_1 \oplus c_3$;
39  $q_4 \leftarrow d_8 \oplus q_0$;
40  $e_4 \leftarrow q_3 \oplus q_4$;
41  $q_5 \leftarrow d_5 \oplus d_2$;
42  $q_6 \leftarrow c_0 \oplus q_5$;
43  $q_7 \leftarrow d_4 \oplus c_1$;
44  $q_8 \leftarrow d_3 \oplus c_0$;
45  $e_0 \leftarrow q_2 \oplus q_6$;
46  $q_9 \leftarrow q_5 \oplus q_7$;
47  $q_{10} \leftarrow d_0 \oplus c_2$;
48  $q_{11} \leftarrow d_7 \oplus d_8$;
49  $e_1 \leftarrow q_4 \oplus q_9$;
50  $q_{12} \leftarrow d_3 \oplus q_{10}$;
51  $e_7 \leftarrow q_3 \oplus q_9$;
52  $e_3 \leftarrow q_2 \oplus q_{12}$;
53  $e_6 \leftarrow q_6 \oplus q_{12}$;
54  $q_{13} \leftarrow q_3 \oplus q_{10}$;
55  $e_5 \leftarrow q_{11} \oplus q_{13}$;
56  $q_{14} \leftarrow q_7 \oplus q_8$;
57  $e_8 \leftarrow q_{13} \oplus q_{14}$;
58  $e_2 \leftarrow q_{11} \oplus q_{14}$;
    // $N_1$ layer
59  $f_0 \leftarrow a_0 \wedge e_0$;
60  $f_1 \leftarrow a_1 \wedge e_1$;
61  $f_2 \leftarrow a_2 \wedge e_2$;
62  $f_3 \leftarrow a_3 \wedge e_3$;
63  $f_4 \leftarrow a_4 \wedge e_4$;
64  $f_5 \leftarrow a_5 \wedge e_5$;
65  $f_6 \leftarrow a_6 \wedge e_6$;
66  $f_7 \leftarrow a_7 \wedge e_7$;
67  $f_8 \leftarrow a_8 \wedge e_8$;
68  $g_0 \leftarrow x_0 \wedge e_0$;
69  $g_1 \leftarrow b_1 \wedge e_1$;
70  $g_2 \leftarrow b_2 \wedge e_2$;
71  $g_3 \leftarrow b_3 \wedge e_3$;
72  $g_4 \leftarrow b_4 \wedge e_4$;
73  $g_5 \leftarrow b_5 \wedge e_5$;
74  $g_6 \leftarrow b_6 \wedge e_6$;
75  $g_7 \leftarrow b_7 \wedge e_7$;
76  $g_8 \leftarrow b_8 \wedge e_8$;
77  $h_0 \leftarrow e_0 \wedge e_3$;
78  $h_1 \leftarrow e_1 \wedge e_4$;
79  $h_2 \leftarrow e_2 \wedge e_5$;
    // $L_2^S$ layer
80  $r_0 \leftarrow f_1 \oplus f_8$;
81  $r_1 \leftarrow f_0 \oplus f_7$;
82  $r_2 \leftarrow f_2 \oplus f_6$;
83  $i_0 \leftarrow r_1 \oplus r_2$;
84  $r_3 \leftarrow f_4 \oplus f_8$;
85  $i_1 \leftarrow r_0 \oplus r_2$;
86  $r_4 \leftarrow f_3 \oplus f_7$;
87  $i_5 \leftarrow r_3 \oplus r_4$;
88  $i_2 \leftarrow r_0 \oplus r_1$;
89  $r_5 \leftarrow f_5 \oplus f_6$;
90  $i_4 \leftarrow r_3 \oplus r_5$;
91  $i_3 \leftarrow r_4 \oplus r_5$;
    // $L_2^S$ layer
92  $s_0 \leftarrow g_1 \oplus g_8$;
93  $s_1 \leftarrow g_0 \oplus g_7$;
94  $s_2 \leftarrow g_2 \oplus g_6$;
95  $j_0 \leftarrow s_1 \oplus s_2$;
96  $s_3 \leftarrow g_4 \oplus g_8$;
97  $j_1 \leftarrow s_0 \oplus s_2$;
98  $s_4 \leftarrow g_3 \oplus g_7$;
99  $j_5 \leftarrow s_3 \oplus s_4$;
100 $j_2 \leftarrow s_0 \oplus s_1$;
101 $s_5 \leftarrow g_5 \oplus g_6$;
102 $j_4 \leftarrow s_3 \oplus s_5$;
103 $j_3 \leftarrow s_4 \oplus s_5$;
    // $L_2^M$ layer
104 $t_0 \leftarrow e_7 \oplus h_1$;
105 $k_0 \leftarrow h_2 \oplus t_0$;
106 $t_1 \leftarrow e_8 \oplus h_0$;
107 $k_2 \leftarrow t_0 \oplus t_1$;
108 $k_1 \leftarrow h_2 \oplus t_1$;
    // $N_2$ layer
109 $l_0 \leftarrow k_0 \wedge i_0$;
110 $l_1 \leftarrow k_1 \wedge i_1$;
111 $l_2 \leftarrow k_2 \wedge i_2$;
112 $l_3 \leftarrow k_0 \wedge i_3$;
113 $l_4 \leftarrow k_1 \wedge i_4$;
114 $l_5 \leftarrow k_2 \wedge i_5$;
115 $m_0 \leftarrow k_0 \wedge j_0$;
116 $m_1 \leftarrow k_1 \wedge j_1$;
117 $m_2 \leftarrow k_2 \wedge j_2$;
118 $m_3 \leftarrow k_0 \wedge j_3$;
119 $m_4 \leftarrow k_1 \wedge j_4$;
120 $m_5 \leftarrow k_2 \wedge j_5$;
    // $L_3$ layer
121 $u_0 \leftarrow l_0 \oplus l_1$;
122 $u_1 \leftarrow m_0 \oplus m_2$;
123 $u_2 \leftarrow m_4 \oplus m_5$;
124 $u_3 \leftarrow l_3 \oplus l_5$;
125 $u_4 \leftarrow l_1 \oplus l_2$;
126 $u_5 \leftarrow m_1 \oplus m_2$;
127 $u_6 \leftarrow m_3 \oplus m_4$;
128 $u_9 \leftarrow l_4 \oplus l_5$;
129 $u_7 \leftarrow u_0 \oplus u_3$;
130 $u_8 \leftarrow u_1 \oplus u_2$;
131 $u_{10} \leftarrow u_1 \oplus u_6$;
132 $y_7 \leftarrow u_2 \oplus u_4$;
133 $y_6 \leftarrow u_4 \ominus u_5$;
134 $y_5 \leftarrow u_1 \ominus u_3$;
135 $y_4 \leftarrow u_5 \oplus y_7$;
136 $y_3 \leftarrow y_6 \ominus u_{10}$;
137 $y_2 \leftarrow u_7 \oplus u_8$;
138 $y_1 \leftarrow u_6 \ominus u_9$;
139 $y_0 \leftarrow u_8 \oplus y_1$;
```

disproves the conjecture made by Boyar and Peralta [BP12] stating that small designs with a gate depth less than 16 are unlikely. In fact, our shallowest design has a gate depth of 14 and is only 11 gates larger than their design of 128 gates and a gate depth of 16, while using the same gate basis of AND, XOR and XNOR. We show this circuit in Algorithm 6.

B.1 Optimizing L_3

L_3 is the last linear layer of the S-Box, separated from the others by the last layer of AND gates, and maps two 6-bit inputs to the 8-bit output of the S-Box. Optimizing with the greedy method proposed by Boyar and Peralta *et al.* [BP10] yields a deep circuit with 19 gates. Applying a SAT-based search yields no circuits with arbitrary gate depth

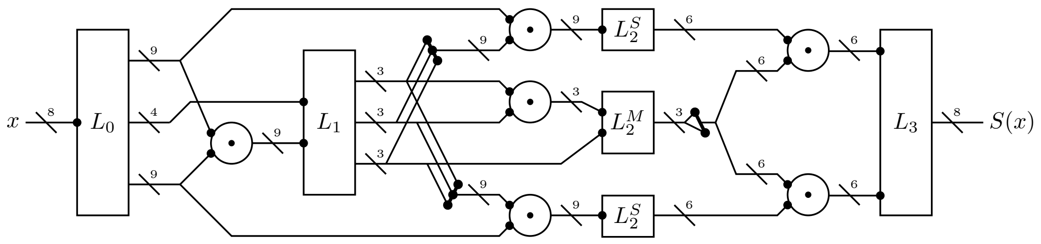


Figure 5: Structure of the AES S-Box with AND-depth of three. The $\odot$ operations represent a pointwise $n \times n$ bit multiplication and consist of n AND gates. The linear layers L_0, L_1, L_2^S, L_2^M and L_3 only contain XOR and XNOR gates.

and only 18 gates, but also does not disprove the possibility of such a circuit within a reasonable time. Therefore, we conjecture that L_3 needs at least 19 linear gates, and continue optimizing the gate depth. Since each output is the linear combination of at most six inputs, we know that the best possible gate depth is three. We ran the optimization method where we declared all inputs to have the same depth, and found a circuit with 19 gates and the optimal gate depth of three.

B.2 Optimizing L_2^S and L_2^M

Similar to L_3, all the inputs of the linear layer L_2^S are outputs of AND gates, so it does not interact with other linear layers, barring the possibility of cross-layer local optimizations.

The naive implementation of L_2^S, based on the bit-level implementations of $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^2}$ multiplications, requires 13 XOR gates and has a gate depth of four. SAT-based optimization yields a smaller circuit with only 12 XOR gates, and proves that no circuits with less gates are possible. Moreover, the optimal gate depth is two, and is indeed realizable with only 12 XOR gates.

Layer L_2^M is parallel to L_2^S and has two 3-bit inputs, one of which comes from L_1. We have found a circuit with 5 XOR gates and a gate depth of two, both of which are provably optimal. Since there is a circuit with both optimal size and gate depth, and the 3-bit input coming from L_1 is also used elsewhere, there seem to be no feasible tradeoffs across the linear layer boundary to L_1.

B.3 Optimizing L_0

L_0 is the first linear layer and has been extensively studied by Boyar *et al.* [BP10, BP12]. They have found an implementation with a gate depth of seven and only 23 gates, as well as an implementation with 27 gates and a gate depth of three. However, while a circuit for L_0 by itself has an optimal gate depth of three, this is not strictly necessary for an AES S-Box with low gate depth. The 4-bit output that goes to L_1 can be relaxed to a depth of four, because the 9-bit input to L_1 has an optimal depth of four due to the AND gates that produce it. We have found a 25 gate circuit, where the two 9-bit outputs of L_0 have a gate depth of three, and the 4-bit output has a gate depth of four.

B.4 Optimizing L_1

Layer L_1 receives a 9-bit input and a 4-bit input. Assuming that they have the same depth, we optimized for the number of gates, and found circuits with 19 XOR gates and no solutions with 18 gates within a reasonable time, conjecturing that there are none. With only 19 XOR gates, the best achievable gate depth seems to be four. Moreover, since the outputs are a combination of at most eight inputs, a gate depth of three is achievable, and we have found one such circuit with only 24 XOR gates.