IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 2, 26 pages.

https://doi.org/10.62056/aee0zoja5

Provably Secure and Area-Efficient Modular Addition over Boolean Shares

Guilhèm Assael[1,2] and Philippe Elbaz-Vincent[2]

[1] STMicroelectronics, Rousset, France
[2] Univ. Grenoble Alpes, CNRS, IF, Grenoble, France

Abstract. Several cryptographic schemes, including lattice-based cryptography and the SHA-2 family of hash functions, involve both integer arithmetic and Boolean logic. Each of these classes of operations, considered separately, can be efficiently implemented under the masking countermeasure when resistance against vertical attacks is required. However, protecting interleaved arithmetic and logic operations is much more expensive, requiring either additional masking conversions to switch between masking schemes, or implementing arithmetic functions as nonlinear operations over a Boolean masking. Both solutions can be achieved by providing masked arithmetic addition over Boolean shares, which is an operation with relatively long latency and usually high area utilization in hardware. A further complication arises when the arithmetic performed by the scheme is over a prime modulus, which is common in lattice-based cryptography. In this work, we propose a first-order masked implementation of arithmetic addition over Boolean shares occupying a very small area, while still having reasonable latency. Our proposal is specifically tuned for efficient addition and subtraction modulo an arbitrary integer, but it can also be configured at runtime for power-of-two arithmetic. To the best of our knowledge, we propose the first such construction whose security is formally proven in the glitch+transition-robust probing model.

Keywords: Secure addition over Boolean shares · Robust probing · Area-efficient masking · Lattice-based cryptography

1 Introduction

Cryptographic schemes are designed with the primary objective of being secure in a black-box model, which assumes that an attacker has no information about the data processed internally by the scheme and only knows its output and/or input data. However, this constraint is rarely met, and attackers can often exploit additional information, such as the time it takes to complete an operation or the power consumption of the device under attack. These methods, known as *side-channel attacks* [KSWH98], can easily allow an attacker to recover a secret key unless countermeasures are explicitly taken against these attacks.

In particular, vertical side-channel attacks consist in observing multiple cryptographic operations on the same secret data and combining the information contained in these traces to recover the secret or part of it. A well-studied and provably secure way to counter this class of side-channel attacks is the *masking*, or *secret-sharing* countermeasure [CJRR99, PR13], which consists in splitting each secret information into several pieces, known as *shares*, randomly chosen so that learning a subset of the shares does not reveal any information about the secret, but the computation can still be performed on the entire set of shares.

The first formal basis for evaluating the security of these circuits, the *t-threshold probing model*, was introduced by Ishai, Sahai and Wagner [ISW03]. An important limitation of this

E-mail: guilhem.assael@st.com (Guilhèm Assael), philippe.elbaz-vincent@math.cnrs.fr (Philippe Elbaz-Vincent)

Received: 2024-04-08 Accepted: 2024-06-03

notion was that it did not take into consideration the effects of *glitches* and *transitions*, two imperfections of physical circuits that made the model insufficient in practice. Their work was consequently extended by Faust *et al.* [FGP+18] to account for this kind of defects, giving the *robust-probing model*. Other leakage models such as the *noisy-leakage* model [CJRR99, PR13], which describes side-channel leakage as a noisy function of the processed secrets, are closer to practice but more difficult to study theoretically. The work of Duc *et al.* [DDF14], among others, bridges the gap by providing a reduction from the threshold-probing model to the noisy-leakage model, a result extended upon by Cassiers *et al.* [CFOS21]. Since the practical relevance of the threshold-probing model is thus being established in its robust version, several works have proposed incremental security notions to allow for the secure construction of complex masked circuits from smaller components (so-called *gadgets*) using a composition approach: in particular, the work of Cassiers and Standaert [CS21] which proposes trivial composability in the glitch+transition-robust probing model.

In recent years, the field of lattice-based cryptography has regained interest since it allows to construct cryptographic schemes that are resistant to quantum computers, while this property is not satisfied by the currently used asymmetric cryptography. In the context of lattice-based cryptography, many types of operations are used within a cryptographic algorithm, that require different masking schemes for an efficient and secure implementation [FBR+22, BGR+21]. Consequently, secure conversions between these masking schemes are a prerequisite for secure implementations of these algorithms [FBR+22]. In hardware implementations, these conversions are costly in terms of area, latency, or both: resource sharing is therefore critical for low-cost implementations. Reusing hardware in this context has been partially shown by Fritzmann *et al.* [FBR+22], but they valued high performance over low area, which is undesirable in the context of embedded systems.

Our contribution We describe an area-efficient masked circuit computing secure modular addition over Boolean shares. The proposed hardware gadget can be configured at runtime for addition or subtraction, either modulo a publicly known arbitrary integer or modulo a power of two. We prove first-order security for this construction in a robust-probing model accounting for glitches and transitions. Then, we synthesize it as an application-specific integrated circuit (ASIC), and show that it exhibits no leakage in simulation.

Outline We start by introducing in Section 2 the background underlying this work. In Section 3, we describe our proposed construction and formally prove its security, and we highlight in Section 4 how our work compares with the state of the art. We then perform a leakage assessment on simulated executions of our design in Section 5, and conclude in Section 6.

2 Preliminaries

2.1 Masking

Masking, also formerly known as *secret sharing* [CJRR99, PR13], is a countermeasure to vertical side-channel attacks that consists in splitting a secret into two or more shares such that no incomplete set of these shares holds any information on the secret, but all shares can be combined into the secret. More formally, given a secret element x of a finite group $(G, \star)$, a sharing of x is a set of d elements $x_0, \ldots, x_{d-1}$ of G such that $x_0 \star \cdots \star x_{d-1} = x$.

The choice of the group is what we call a *masking scheme*. In this work, we speak of Boolean masking when the shares are elements of $(\mathbb{F}_2^n, \oplus)$ for some integer n, of power-of-two arithmetic masking when they are elements of $(\mathbb{Z}/2^n\mathbb{Z}, +)$, and of modulo-$q$ arithmetic masking when they are elements of $(\mathbb{Z}/q\mathbb{Z}, +)$ for some arbitrary integer q.

2.2 Notations and terminology

For a quantity represented by shares $a_0, \ldots, a_{d-1}$, the set of all shares is represented by a_*. The symbol without any share index, a, stands for the unmasked value: for Boolean sharing, $a = \bigoplus_i a_i$. For a set $S \subset \mathbb{F}_2^n$ (where $\mathbb{F}_2$ is the field with two elements), we denote by $\mathcal{B}_2(S) = \{(x, y) \in \mathbb{F}_2^{2n} \mid x \oplus y \in S\}$ the set of Boolean sharings of elements of S.

Since our equations mix bitwise logic operations with arithmetic computations, we employ the following notation: Boolean XOR (sum in $\mathbb{F}_2$) is denoted by $\oplus$, Boolean AND (product in $\mathbb{F}_2$) is denoted by $\cdot$ or juxtaposition; the one's complement (Boolean negation) of a is $\overline{a}$. In contrast, $+$ and $-$ express arithmetic addition and subtraction in $\mathbb{Z}$. Arithmetic operations on bit strings assume a binary representation using two's complement for negative numbers, and Boolean operations are applied bitwise. As usual, Boolean AND has precedence over Boolean XOR unless parentheses are used for grouping.

Consider two bit strings $s \in \mathbb{F}_2^m$ and $t \in \mathbb{F}_2^n$. For $0 \leqslant i \leqslant m-1$, we denote by $s[i]$ the ith component of s. We represent by $s \parallel t = u$ the concatenation of s and t, that is, $u[i] = t[i]$ if $0 \leqslant i < n$ and $u[i] = s[i-n]$ if $n \leqslant i < n+m$. Furthermore, $s \gg p = v \in \mathbb{F}_2^{m-p}$ is s right shifted p times, that is, $v[i] = s[i+p]$ for $0 \leqslant i < m-p$. Note that bit strings are considered in big-endian order, with higher indexes on the left.

If S is a set, $a \leftarrow S$ denotes sampling uniformly at random the value of a from set S. We denote by $\mathrm{Reg}[\cdot]$ a register, which is a sequential gate delaying its input by one clock cycle; given a shared value $x_* \in \mathcal{B}_2(\mathbb{F}_2)$ and a random bit $r \in \mathbb{F}_2$, we define $\mathrm{Refresh}(x_*, r) = (x_0 \oplus r, x_1 \oplus r)$.

2.3 Security model

We want to formally prove the security of our constructions in a probing model that represents as closely as possible the behavior of ASIC implementations of secure hardware, in particular by including the effects of *glitches* and *transitions*. The former refers to the progressive and uneven propagation of values across combinational logic, which causes the gates to switch several times before reaching their final value, potentially leaking secrets [FG05]. The latter reflects that logic gates and wires leak depending not only their logic level, but also on their switching, in particular over successive clock cycles, which can also cause vulnerabilities in masked implementations. We thus consider both hardware defects, through the attacker model of *glitch+transition-robust probing* [CS21].

Since the direct security analysis of complex masked circuits is often infeasible, formal security proofs usually rely on composability notions, where the overall circuit is split into smaller individual *gadgets* for which the security properties are easier to prove. To allow for unrestricted gadget composition in the glitch+transition-robust probing model, Cassiers and Standaert introduced the Output Probe-Isolating Non-Interference (O-PINI) notion [CS21], which is a stronger evolution of their earlier PINI notion [CS20]. We recall the O-PINI notion in the specific context of our work, that is, for gadgets having two shares and thus only targeting first-order security.

Definition 1 (Output Probe-Isolating Non-Interference [CS21, Definition 20 with $t = 1$])**.** A gadget G with 2 shares is O-PINI if and only if for any probe I_1 on its internal wires, there exists a share index i so that the observations corresponding to I_1 and probes on all output shares of index i can be simulated using only the input shares with index i.

In the glitch+transition-robust probing model, each of the probes mentioned in Definition 1 must be extended across both glitches and transitions within the considered gadget. A glitch-extended probe on a net leaks the value of all combinational inputs contributing to the value of the net. A transition-extended probe on a net leaks both its current value and its value at the previous clock cycle. The combination of the two is done by first transition-extending each probe, then glitch-extending the resulting set of probes.

The circuit model of Cassiers and Standaert [CS21] is based on the one of Ishai *et al.* [ISW03], with added notions that allow for reusing physical gates to implement different logical functions across clock cycles. We only give a high-level overview of this model here, and refer the reader to [CS21] for formal definitions. At the core of this model are the notions of *structural gates* and *structural wires*, which describe the physical configuration of the circuit, including the latency of sequential gates (flip-flops), as a directed graph. On top of this physical view of a circuit, a logical one describes its behavior over time: a *circuit execution* consists of replications of the gates of a structural circuit at each clock cycle, with wires that connect these replicas according to the latency of the involved gates.

The notion of gadget in the model of Cassiers and Standaert is twofold: on one hand, a *gadget execution* is a subset of the gates and wires of a circuit execution. Those wires whose source is not included in the gadget are referred to as its *inputs*, and are partitioned into tuples of d elements, d being the number of shares of the gadget. A gadget furthermore has a set of *outputs* (taken from the outputs of its constituting gates), likewise partitioned into sets of d shares. Disjoint gadget executions can be composed by linking outputs to inputs, respecting the order of shares and ensuring that the composition graph contains no cycles: the inputs of a gadget execution cannot depend directly or indirectly on one of its outputs.

On the other hand, the notion of *structural gadget* is introduced: it is a set of disjoint gadget executions that are identical except for a translation in time, that is, that all use the same structural gates and wires but at different clock cycles. Distinct structural gadgets must not share any structural gates or wires among them. In the terminology of [CS21], a structural gadget is said to be *pipeline* if its canonical execution uses each of its structural gates and wires only once, which will be the case for all our elementary gadgets.

We note that there exist some automated tools to check the security of masked designs. FullVerif [CGLS21], on one hand, analyzes composite circuits made of gadgets with some known security properties, and checks the global security of the circuit through a composition approach. IronMask [BMRT22] and SILVER [KSM20], on the other hand, analyze the internal construction of gadgets to formally prove their security. However, none of these tools seems to support O-PINI security yet, so they cannot be used to check the glitch+transition-robustness of iterative circuits. This absence of automated tools for our purpose is not a concern, since we prove the security of our individual gadgets by hand, according to a security notion that allows for trivial composition.

2.4 Lattice-based cryptography

Lattice-based cryptography, a long-standing class of cryptographic schemes relying on hard mathematical problems over lattices, has gained widespread interest in recent years in the context of the development of post-quantum cryptography. Most notably, the US National Institute of Standards and Technology (NIST) has initiated the standardization of two lattice-based cryptography schemes, proposed under the names CRYSTALS-Kyber [SAB+20] and CRYSTALS-Dilithium [LDK+20], to be respectively standardized as ML-KEM [FIP23a] and ML-DSA [FIP23b]. An important aspect of these schemes is that they embed Boolean logic and modular arithmetic operations, both of which must be implemented securely when side-channel attacks are a concern. Satisfying this constraint usually requires the implementation of secure operations for the conversion between Boolean and arithmetic masking. It has been shown, among others, by Fritzmann *et al.* [FBR+22] that masking conversions based on secure addition over Boolean shares (SecAdd [CGV14]) offer ideal versatility and resource efficiency. Precisely, SecAdd and its extension to modular addition proposed by Barthe *et al.* [BBE+18], allow to perform both Boolean-to-Arithmetic and Arithmetic-to-Boolean conversion, where the arithmetic masking may be either modulo a power of two or modulo a prime. This flexibility is very welcome when implementing side-channel–resistant lattice-based cryptography on embedded devices.

2.5 Binary-addition algorithms

We briefly recall the main architectures for binary addition and highlight their characteristics. The most basic architecture, the ripple carry adder [Mac61], is built from a chain of n full adders (n being the number of bits of the summands): each, given as input one bit of each summand and an input carry, computes an output bit and an output carry. The full adders are chained, from least to most significant, so that each sends its output carry to the next full adder. The main drawback of this architecture is its long propagation delay, since the input carry ripples through n successive full adders before the sum is complete.

To speed up the propagation of the carry, a carry-select adder [Bed62] can be used: the summands are divided into groups of a smaller width, and two sums are computed for each group of bits: one assuming that the group input carry is set, the other assuming that it is cleared. Then, depending on the actual input carry, the correct sum and output carry are selected for each block. The carry-skip or carry-bypass adder [LB61] similarly divides its input width into groups, but it computes a single sum for each group, once the input carry is available. Only the carry propagation from one block to the next is sped up, thanks to the precomputation of *carry-skip* and *carry-generate* signals for each block.

While the three above architectures have linear latency, addition can also be computed in logarithmic time, using a parallel-prefix adder: this architecture arranges carry-lookahead logic in a tree of logarithmic depth to quickly propagate the carries over groups of increasing size [Skl60]. The drawback of this construction is its larger area, which grows in $O(n \log(n))$.

In unprotected implementations performing addition in a single cycle, it is clear that the ripple-carry adder requires the smallest area and highest latency, while parallel-prefix adders are among the largest and fastest [Mac61, LB61, WT90]. By configuring the size of groups, a wide range of intermediate performances can be obtained from carry-skip, carry-select, and similar architectures [Mac61, Bed62].

While generalizing this comparison to masked implementations is difficult, the available literature confirms a similar trend in terms of area and latency, with slow but small masked ripple carry adders and large but fast masked parallel-prefix adders (Table 7, Table 8). We are not aware of any masked implementation of carry-select or carry-skip adders. Previous works [SMG15, FBR+22, BG22, CGM+23] have shown that fully pipelined parallel-prefix adders require a very large area, and converting them to iterative designs while keeping resistance against glitches and transitions would require switching to iterated glitch+transition-robust gadgets, which have higher area and latency than gadgets without this property [CS21, KM22]. We thus do not expect that iterative implementations of parallel-prefix adders can reach a sufficiently low area to be relevant in resource-constrained implementations. This explains why we specifically investigate the ripple-carry adder, whose specific advantages in the case of modular addition are discussed in Subsection 3.3.

3 Secure modular arithmetic over Boolean shares

In this section, we describe the construction of our masked circuit for secure addition and subtraction. We first recall how modular addition can be performed using regular addition followed by trial subtraction, and then describe how ripple-carry addition works. We then introduce the nonlinear gadgets we use to implement this operation, prove their security in the robust-probing model, and show exactly how they can be assembled into masked modular addition. We briefly describe how related operations can be implemented with the same circuit: modular subtraction, and both addition and subtraction modulo a power of two. Finally, we recall how secure addition and subtraction can be used as the main tool to implement conversions between Boolean and arithmetic masking.

3.1 Modular addition using trial subtraction

Given an integer $q < 2^n$, we can compute the sum of two integers $a, b \in [\![0, q-1]\!]$ modulo q by performing the sum without modular reduction, then subtracting q to attempt modular reduction, and selecting which of these two results is valid based on the sign in the output of the subtraction, as described in Equation 1:

$$(a+b) \bmod q = \begin{cases} (a+b)-q & \text{if } a+b-q \geqslant 0, \\ (a+b) & \text{otherwise.} \end{cases} \tag{1}$$

When using an n-bit adder with an output carry, the full precision of addition $a+b$ can be kept, but it is not the case for the subtraction of q since it would require subtracting from an $(n+1)$-bit quantity. It is however possible to express this condition in a different way. Since $a+b < 2q < 2^n+q$, modular reduction must be performed exactly when either of the two mutually exclusive conditions in Equation 2 is true:

$$a+b \geqslant 2^n \quad \text{or} \quad ((a+b) \bmod 2^n) + (2^n - q) \geqslant 2^n. \tag{2}$$

Modular subtraction is similar, except that the condition for selecting between the two results is known directly after the first subtraction. If the carry is cleared, which happens when the result is negative, then a modular reduction has to be performed by adding q to the difference, as in Equation 3:

$$(a-b) \bmod q = \begin{cases} a-b & \text{if } a-b \geqslant 0, \\ a-b+q & \text{otherwise.} \end{cases} \tag{3}$$

3.2 Secure ripple-carry addition

We now describe the algorithm that we use for secure modular addition: as discussed previously, we use ripple-carry addition since it best fits our aim of low area utilization. We show in Algorithm 1 how ripple-carry addition is performed, based on shift registers to rotate the operands by one bit at a time. An output carry is provided by the operation.

Algorithm 1: Ripple-carry adder over n bits

Input: augend $\in \mathbb{F}_2^n$, addend $\in \mathbb{F}_2^n$
Output: sum $\in \mathbb{F}_2^n$, $c \in \mathbb{F}_2$ such that $c \,\|\, \text{sum} = \text{augend} + \text{addend}$

```
sum = 0 ∈ F_2^n,      c = 0 ∈ F_2
for i = 0 to n − 1 do
    z ‖ s = augend[0] + addend[0] + c ∈ F_2^2
    sum = s ‖ (sum ≫ 1)
    augend = 0 ‖ (augend ≫ 1)
    addend = 0 ‖ (addend ≫ 1)
    c = z
end
return sum, c
```

Given the nature of carry propagation, repeated summation (accumulation) operations can be intermeshed: in the case of modular addition where quantities $a+b$ (which we call *raw sum*) and $(a+b)+(2^n-q)$ (named *offsetted sum*) must be computed, the second quantity can be computed simultaneously with the first, without waiting for the first carry to propagate. Secondly, since shifting the operands right at each cycle frees their most significant bit, the freed positions can store the newly computed bits of the sums. Finally, the choice in Equation 1 can be implemented by computing the raw sum $(a+b)$ and the bit-wise difference between the offsetted and raw sums $((a+b) \oplus (a+b-q))$, and optionally adding the latter to the former. These transformations give Algorithm 2, which computes two simultaneous ripple-carry additions and uses their output carries to perform the modular reduction.

Algorithm 2: Addition modulo q using n-bit ripple-carry adders

Input: augend $\in \mathbb{F}_2^n$, addend $\in \mathbb{F}_2^n$
Parameters: Modulus $q < 2^n$
Output: sum $\in \mathbb{F}_2^n$ such that sum = (augend + addend) mod q

1 $c, d = 0, 0$
2 **for** $i = 0$ *to* $n - 1$ **do**
3 $\quad z \parallel s = \text{augend}[0] + \text{addend}[0] + c \qquad \in \mathbb{F}_2^2$
4 $\quad \xi \parallel \rho = s + (2^n - q)[i] + d \qquad \in \mathbb{F}_2^2$
5 $\quad \text{augend} = s \parallel (\text{augend} \gg 1)$ *// Shift augend and save new raw-sum bit*
6 $\quad \text{addend} = (\rho \oplus s) \parallel (\text{addend} \gg 1)$ *// Save difference between raw and offsetted sum bits*
7 $\quad c, d = z, \xi$ *// Use the new carries as input for next iteration*
8 **end**
9 **return** augend $\oplus$ (addend ***if*** $c \oplus d$ ***else*** 0)

3.3 Secure modular addition over Boolean shares

By implementing all operations of Algorithm 2 using masked gadgets over Boolean sharings, we can compute modular addition securely over Boolean-masked values. This construction is shown in Algorithm 3, where we assume the presence of secure gadgets SDFA_m, which securely computes the sum and carry bits at lines 3–4 of Algorithm 2 (where parameter m successively holds the bits of $2^n - q$ from least to most significant), and SMx_n, which outputs its first operand, conditionally XORed with its second operand (each have n bits) depending on the value of the third (a Boolean sharing of a single bit).

Algorithm 3: Secure modular addition over Boolean sharings

Input: $\text{augend}_* \in \mathcal{B}_2(\mathbb{F}_2^n)$, $\text{addend}_* \in \mathcal{B}_2(\mathbb{F}_2^n)$
Parameters: $n \in \mathbb{N}$, modulus $q \in [\![1, 2^n]\!]$ where 2^n is represented as the all-0 bit string
Output: $\text{sum}_* \in \mathcal{B}_2(\mathbb{F}_2^n)$ such that sum = (augend + addend) mod q

1 $c_* = 0 \in \mathcal{B}_2(\mathbb{F}_2)$ *// Initial carry for raw sum*
2 $d_* = 0 \in \mathcal{B}_2(\mathbb{F}_2)$ *// Initial carry for offsetted sum*
3 **for** $i = 0$ *to* $n - 1$ **do**
4 $\quad m = (2^n - q)[i]$
$\quad$ *// Compute the sum and carry bits for the raw and offsetted sums*
5 $\quad s_*, z_*, \delta_*, \xi_* = \text{SDFA}_m(\text{augend}_*[0], \text{addend}_*[0], c_*, d_*)$
$\quad$ *// Rotate the operand registers and store the sum bits at their top*
6 $\quad \text{augend}_* = s_* \parallel (\text{augend}_* \gg 1)$
7 $\quad \text{addend}_* = \delta_* \parallel (\text{addend}_* \gg 1)$ *// δ corresponds to $\rho \oplus s$ in Algorithm 2*
8 $\quad c_*, d_* = z_*, \xi_*$ *// Use the new carries as input carries for next iteration*
9 **end**
10 $e_* = c_* \oplus d_* \in \mathcal{B}_2(\mathbb{F}_2)$
11 **return** $\text{SMx}_n(\text{augend}_*, \text{addend}_*, e_*)$

Fritzmann *et al.* [FBR+22] use a different approach for secure modular addition: by assuming that modulus q has already been subtracted from one of the summands before secure addition, the carry output by the first addition already indicates whether modular reduction should be performed. The second secure addition then conditionally applies this reduction by adding either q or 0. This method is more efficient in their work since it avoids the need for a secure multiplexer. However, the two computationally expensive secure additions remain, and they can no longer be computed in parallel since the second depends on the output carry of the first. That method would thus be highly suboptimal in our setting.

In Algorithm 3, we have intentionally hidden the latency of operations for better clarity. This latency will be made explicit in Subsubsection 3.3.3. We now describe the inner gadgets of secure modular addition, and prove their security in the robust-probing model.

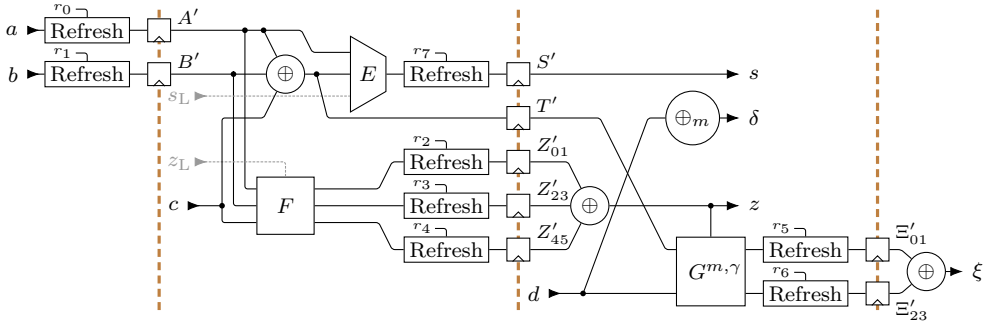

Figure 1: The SecDualFullAdder gadget. Given masked inputs a, b, input carry c, and input offsetted carry d, the SDFA operation of the gadget securely computes raw-sum bit s, offset bit δ, output raw carry bit z, and output offsetted carry ξ. Each edge is a 2-bit bus. Node $\oplus_m$ indicates Boolean addition of m (the ith bit of the negated modulus) to the first share.

Algorithm 4: SecDualFullAdder

Input: Shares $a_*, b_* \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 0, shares $c_*, s_{\mathrm{L},*}, z_{\mathrm{L},*} \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 1, shares $d_* \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 2

Parameters: $m, \gamma \in \mathbb{F}_2$, $E = (E_0, E_1) \in (\mathbb{F}_2^4 \to \mathbb{F}_2)^2$,
$F = (F_0, \ldots, F_5) \in (\mathbb{F}_2^4 \to \mathbb{F}_2)^2 \times (\mathbb{F}_2^3 \to \mathbb{F}_2)^2 \times (\mathbb{F}_2^2 \to \mathbb{F}_2)^2$,
$G^{m,\gamma} = (G_0^{m,\gamma}, \ldots, G_3^{m,\gamma}) \in (\mathbb{F}_2^3 \to \mathbb{F}_2)^2 \times (\mathbb{F}_2^2 \to \mathbb{F}_2)^2$

Input randomness: $r_0, \ldots, r_7 \in \mathbb{F}_2$

Output: Shares $s_*, z_*, \delta_* \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 2 and shares $\xi_* \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 3, such that $s = E(a, a \oplus b \oplus c, s_{\mathrm{L}})$, $z = F(A, B, c, z_{\mathrm{L}})$, $\delta = d \oplus m$, $\xi = G^{m,\gamma}(s, d, z)$

1 $A_* = \mathrm{Reg}[\mathrm{Refresh}(a_*, r_0)]$
2 $B_* = \mathrm{Reg}[\mathrm{Refresh}(b_*, r_1)]$
3 $T_* = A_* \oplus B_* \oplus c_*$
4 $S_* = E_0(A_0, T_0, s_{\mathrm{L},0}), E_1(A_1, T_1, s_{\mathrm{L},1})$
5 $Z_0, Z_1 = F_0(A_0, B_0, c_0, z_{\mathrm{L},0}), F_1(A_1, B_1, c_1, z_{\mathrm{L},1})$
6 $Z_2, Z_3 = F_2(A_1, B_0, c_0), F_3(A_0, B_1, c_1)$
7 $Z_4, Z_5 = F_4(B_1, c_0), F_5(B_0, c_1)$
8 $T'_* = \mathrm{Reg}[T_*]$
9 $s_* = \mathrm{Reg}[\mathrm{Refresh}(S_*, r_7)]$
10 $z_* = \mathrm{Reg}[\mathrm{Refresh}((Z_0, Z_1), r_2)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_2, Z_3), r_3)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_4, Z_5), r_4)]$
11 $\delta_* = d_* \oplus (m, 0)$
12 $\Xi_0, \Xi_1 = G_0^{m,\gamma}(T'_0, d_0, z_0), G_1^{m,\gamma}(T'_1, d_1, z_1)$
13 $\Xi_2, \Xi_3 = G_2^{m,\gamma}(T'_1, d_0), G_3^{m,\gamma}(T'_0, d_1)$
14 $\xi_* = \mathrm{Reg}[\mathrm{Refresh}((\Xi_0, \Xi_1), r_5)] \oplus \mathrm{Reg}[\mathrm{Refresh}((\Xi_2, \Xi_3), r_6)]$
15 **return** $s_*, z_*, \delta_*, \xi_*$

3.3.1 Secure sum and carry computation

We first study SecDualFullAdder, which implements the secure computation of the sum and carry bits at line 5 of Algorithm 3. To get a glitch+transition-robust O-PINI gadget having reasonable latency, we need to integrate all four computations into an atomic gadget. Furthermore, in order to avoid unnecessary duplication of resources, the gadget implements other operations: besides sum and carry computation (operation SDFA), it provides register operations (SDFAin and SDFAcp with one and two cycles of latency respectively) as well as a secure-selection operation (SDFAmx). We show in Figure 1 a schematic representation of the gadget and describe in Algorithm 4 the logic equations that are common to all operations[1], using black-box functions E, F and G, which are

[1]The algorithm listings specialized to each of the four operations of the gadget are given in Appendix A.

Table 1: Unmasked output equations of SecDualFullAdder based on configuration

Output	SDFA	SDFAin	SDFAcp	SDFAmx
s	$a \oplus b \oplus c$	s_{L}	a	s_{L}
z	$a \cdot b \oplus b \cdot c \oplus a \cdot c$	z_{L}	b	$a \oplus b \cdot c$
δ	$d \oplus m$	$d \oplus m$	$d \oplus m$	$d \oplus m$
ξ	$(s \cdot d \oplus s \cdot m \oplus d \cdot m) \oplus \gamma \cdot z$	0	d	0

public parameters. The unmasked computation carried out by each operation is specified in Table 1, with Table 2 fully defining the contents of the black-box functions to achieve it.

One full execution of SDFA needs to spread over four cycles so it can achieve its target security (robust O-PINI). The inputs and outputs are distributed over these cycles to minimize the overall latency of secure modular addition, by presenting a single cycle of latency from the carry inputs (c and d) to the corresponding carry outputs (z and ξ). Initially, sharings of one bit of each summand are provided through the a and b inputs, and immediately refreshed. After a first register barrier, the raw input carry, c, is provided, and the masked computation of the raw sum and carry bits (S, Z) is performed. These are refreshed and registered in the second barrier, after which they are output as s and z. A second, independent sharing of s is stored as T', to be used for the nonlinear computation of the offsetted carry bit, Ξ. This computation also involves the input offsetted carry, d, which is provided at the same cycle. After the third register barrier, the refreshed shares of the offsetted carry, ξ_*, are output.

To simplify the composition diagram, two additional linear computations are integrated within SDFA, although they could be performed externally without affecting the security proofs: the computation of the offset, $\delta = d \oplus m$, and the Boolean addition of the two carries z and ξ in the last execution of SDFA, to decide whether modular reduction should be performed (see line 10). This latter result overwrites the ξ output of SDFA when parameter γ equals 1.

The correctness of this gadget is easily checked by recursively evaluating equations, for instance for output ξ of the SDFA operation:

$$\begin{aligned}\xi_0 \oplus \xi_1 = \bigoplus_{i=0}^{3} \Xi_i &= (T'_0 \oplus T'_1 \oplus d_0 \oplus d_1) \cdot m \oplus (T'_0 \oplus T'_1) \cdot (d_0 \oplus d_1) \oplus \gamma \cdot (z_0 \oplus z_1) \\ &= ((S_0 \oplus S_1) \oplus d) \cdot m \oplus (S_0 \oplus S_1) \cdot d \oplus \gamma \cdot z = (s \cdot m \oplus s \cdot d \oplus d \cdot m) \oplus \gamma \cdot z\end{aligned}$$

which corresponds to taking the carry bit of $s + d + m$, and furthermore adding it with z over $\mathbb{F}_2$ when $\gamma = 1$.

We now prove the security of SecDualFullAdder in the model of [CS21].

Proposition 1. SecDualFullAdder *is glitch-robust O-PINI.*

Proof. We list in Table 3 the glitch-extended probes on output shares of SecDualFullAdder.

Now, consider any of the internal probes listed in the first column of Table 4. Without loss of generality, we will consider the probe in the last row of the table: Ξ'_2, that is, the result of refreshing Ξ_2 with r_6. This probe is of particular interest as its extension involves cross-domain terms, which are the limiting factor to the security of nonlinear gadgets.

The probe glitch-extends to probes on T'_1, d_0, and r_6. We now build a simulator for the extended probe on Ξ'_2 as well as extended probes on outputs having share index 0, from the knowledge of input shares with index 0: a_0, b_0, c_0, d_0, $s_{\mathrm{L},0}$, $z_{\mathrm{L},0}$.

The simulator first samples at random all the values listed in the second column, namely, T'_1, r_6, S'_0, Z'_0, Z'_2, Z'_4 and Ξ'_0. This sampling is indistinguishable from the actual gadget execution since each of these values is blinded with (or is a) fresh random bits, respectively

Table 2: Parameterization of SecDualFullAdder depending on configuration

Internal function	SDFA	SDFAin	SDFAcp	SDFAmx
$E_0(A_0, T_0, s_{\mathrm{L},0})$	T_0	$s_{\mathrm{L},0}$	A_0	$s_{\mathrm{L},0}$
$E_1(A_1, T_1, s_{\mathrm{L},1})$	T_0	$s_{\mathrm{L},1}$	A_1	$s_{\mathrm{L},1}$
$F_0(A_0, B_0, c_0, z_{\mathrm{L},0})$	$A_0 \cdot B_0 \oplus A_0 \cdot c_0 \oplus B_0 \cdot c_0$	$z_{\mathrm{L},0}$	B_0	$A_0 \oplus B_0 \cdot c_0$
$F_1(A_1, B_1, c_1, z_{\mathrm{L},1})$	$A_1 \cdot B_1 \oplus A_1 \cdot c_1 \oplus B_1 \cdot c_1$	$z_{\mathrm{L},1}$	B_1	$A_1 \oplus B_1 \cdot c_1$
$F_2(A_1, B_0, c_0)$	$A_1 \cdot (B_0 \oplus c_0)$	0	0	0
$F_3(A_0, B_1, c_1)$	$A_0 \cdot (B_1 \oplus c_1)$	0	0	0
$F_4(B_1, c_0)$	$B_1 \cdot c_0$	0	0	$B_1 \cdot c_0$
$F_5(B_0, c_1)$	$B_0 \cdot c_1$	0	0	$B_0 \cdot c_1$
$G_0^{m,\gamma}(T_0', d_0, z_0)$	$(T_0'm \oplus d_0m \oplus T_0'd_0) \oplus \gamma \cdot z_0$	0	d_0	0
$G_1^{m,\gamma}(T_1', d_1, z_1)$	$(T_1'm \oplus d_1m \oplus T_1'd_1) \oplus \gamma \cdot z_1$	0	d_1	0
$G_2^{m,\gamma}(T_1', d_0)$	$T_1' \cdot d_0$	0	0	0
$G_3^{m,\gamma}(T_0', d_1)$	$T_0' \cdot d_1$	0	0	0

Table 3: Glitch extension of probes on output shares of SecDualFullAdder. Prime symbols represent the refreshed value of the corresponding quantity, e.g. $\Xi_2' = \Xi_2 \oplus r_6$.

Share index	Glitch-extended probes on output shares			
0	$s_0 = S_0'$,	$z_0 = Z_0' \oplus Z_2' \oplus Z_4'$,	$\delta_0 = d_0 \oplus m$,	$\xi_0 = \Xi_0' \oplus \Xi_2'$
1	$s_1 = S_1'$,	$z_1 = Z_1' \oplus Z_3' \oplus Z_5'$,	$\delta_1 = d_1$,	$\xi_1 = \Xi_1' \oplus \Xi_3'$

Table 4: Simulation of a glitch-extended internal probe of even index in SecDualFullAdder together with all extended probes on output shares having index 0. All input shares with index 0 (a_0, b_0, c_0, d_0, $s_{\mathrm{L},0}$, $z_{\mathrm{L},0}$) are known. Prime symbols represent the refreshed value of the corresponding quantity, e.g. $\Xi_2' = \Xi_2 \oplus r_6$. Public parameters m and γ are known.

Glitch-extended internal probes	Values simulated by random sampling	Notes
$A_0 = a_0 \oplus r_0, B_0 = b_0 \oplus r_1$ $T_0' = T_0 = A_0 \oplus B_0 \oplus c_0$ $S_0' = E_0(A_0, T_0, s_{\mathrm{L},0}) \oplus r_7$	r_0, r_1, r_7 $Z_0', Z_2', Z_4', \Xi_0', \Xi_2'$	
$Z_0' = F_0(A_0, B_0, c_0, z_{\mathrm{L},0}) \oplus r_2$	r_0, r_1, r_2 $S_0', Z_2', Z_4', \Xi_0', \Xi_2'$	Compute $A_0 = a_0 \oplus r_0$, $B_0 = b_0 \oplus r_1$
$Z_2' = F_2(A_1, B_0, c_0) \oplus r_3$	A_1, r_1, r_3 $S_0', Z_0', Z_4', \Xi_0', \Xi_2'$	Compute $B_0 = b_0 \oplus r_1$ A_1 is blinded with r_0 (not probed)
$Z_4' = F_4(B_1, c_0) \oplus r_4$	$B_1, r_4, S_0', Z_0', Z_2', \Xi_0', \Xi_2'$	B_1 is blinded with r_1 (not probed)
$\Xi_0' = G^{m,\gamma}(T_0', d_0) \oplus r_5$	$T_0', r_5, S_0', Z_0', Z_2', Z_4', \Xi_2'$	T_0' is blinded with r_0 and r_1 (not probed)
$\Xi_2' = G^{m,\gamma}(T_1', d_0) \oplus r_6$	$T_1', r_6, S_0', Z_0', Z_2', Z_4', \Xi_0'$	T_1' is blinded with r_0 and r_1 (not probed)

$r_0 \oplus r_1$, r_6, r_7, r_2, r_3, r_4, r_5. Then, Ξ'_2 can be computed from these values and d_0, which is a simulator input. In turn, it is clear from Table 3 that these values are sufficient to compute the glitch-extended probes on outputs having share index 0.

The other lines of Table 4 likewise indicate the values which must be sampled to simulate any other internal probe having even index. Similarly, any internal probe having odd index, together with extended probes on outputs with index 1, can be simulated from the input shares having index 1: the equivalent of Table 4 for odd-index probes is derived by toggling the parity of the index of all quantities in the table except for random bits. □

Proposition 2. SecDualFullAdder *is iterated glitch+transition-robust O-PINI.*

Proof. Since SecDualFullAdder is pipeline and glitch-robust O-PINI, by [CS21, Lemma 2], it is also iterated glitch+transition-robust O-PINI. □

3.3.2 Masked multiplexer and dual-register gadget

The summands must be stored in a shift register in order to be sent bit by bit to SecDualFullAdder; moreover, as discussed earlier, the same shift registers are progressively loaded with the calculated bits of the raw sum and the *offset* (the exclusive-or of the raw and offsetted sums). Furthermore, modular addition and subtraction involve optionally XORing the offset with the raw sum, this selection being performed securely. To save area, we implement both the shift registers and the secure selector using the same physical registers, as two configurations of a single gadget, which we call SecMux_n for n-bit size.

The construction of the SecMux_2 gadget is given in Figure 2 and its internal equations are laid out in Algorithm 5. In the secure-multiplexer configuration (SMx), the gadget takes a Boolean sharing s_* of a single bit, as well as Boolean sharings a_* and b_* of two n-bit quantities at the next cycle, and outputs a sharing of either a or $a \oplus b$ depending on s by computing $a \oplus bs$ (where s is broadcast to all bits of b). In the dual-register configuration (SDR), the gadget refreshes its a and b inputs into output sharings x and y after a one-cycle delay. Input s is ignored, and output z is unused. We highlight that the two configurations of the gadget use the same Boolean operators internally, only with different operands: where SMx operates on the shares of s, SDR instead uses public constants 0 and 1. Thus, they can be implemented using the same set of structural gates, with a public parameter to override the secret inputs with constants when implementing SDR. Proving the security of the gadget in the SMx mode is thus sufficient.

Proposition 3. SecMux *is glitch-robust O-PINI.*

Proof. Let us consider an internal probe on the input of register $y_0[j]$ for some arbitrary (but fixed) $j \in [\![0, n-1]\!]$. As before, this probe is of particular interest because it involves

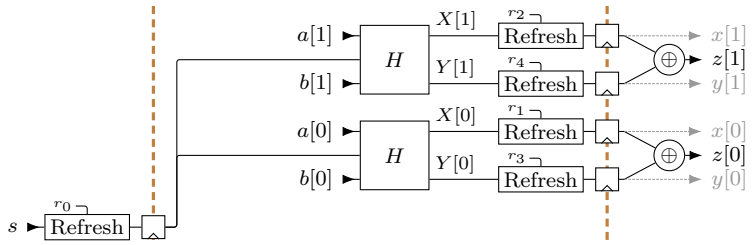

Figure 2: The SecMux_2 gadget over 2-bit values: given Boolean sharings of $a \in \mathbb{F}_2^2$, $b \in \mathbb{F}_2^2$ and $s \in \mathbb{F}_2$, this gadget outputs as $z \in \mathbb{F}_2^2$ a sharing of either a or $a \oplus b$ depending on the value of s. Outputs $x \in \mathbb{F}_2^2$ and $y \in \mathbb{F}_2^2$ are only used when this gadget is configured to implement a dual register instead of secure selection.

Algorithm 5: SecMux_n

Input: Shares $s_* \in \mathcal{B}_2(\mathbb{F}_2)$ with latency 0, shares $a_*, b_* \in \mathcal{B}_2(\mathbb{F}_2^n)$ with latency 1.
Parameters: gadget $\in \{\mathrm{SMx}, \mathrm{SDR}\}$
Input randomness: $r_0, \ldots, r_{2n} \in \mathbb{F}_2$
Output: Shares $x_*, y_*, z_* \in \mathcal{B}_2(\mathbb{F}_2^n)$ with latency 2, s. t. $\begin{cases} \text{gadget} = \mathrm{SMx} \Rightarrow z = a \oplus bs \\ \text{gadget} = \mathrm{SDR} \Rightarrow (x, y) = (a, b) \end{cases}$

$S_* = \mathrm{Reg}[\mathrm{Refresh}(s_*, r_0)]$
if gadget $= \mathrm{SMx}$ **then** $v = 0$ **else** $v = 1$ *// Public override*
for $i = 0$ *to* $n - 1$ **do**
  $X_0[i] = a_0[i] \oplus b_0[i] \cdot S_0 \cdot \overline{v}$
  $X_1[i] = a_1[i] \oplus b_1[i] \cdot S_1 \cdot \overline{v}$
  $Y_0[i] = b_0[i] \cdot (S_1 \mid v)$ *// $\mid$ is Boolean OR*
  $Y_1[i] = b_1[i] \cdot (S_0 \mid v)$
  } $X_*[i], Y_*[i] = H(a_*[i], b_*[i], S_*)$
  $x_*[i] = \mathrm{Reg}[\mathrm{Refresh}(X_*[i], r_{i+1})]$
  $y_*[i] = \mathrm{Reg}[\mathrm{Refresh}(Y_*[i], r_{i+n+1})]$
  $z_*[i] = x_*[i] \oplus y_*[i]$
end
return x_*, y_*, z_*

Table 5: Glitch extension of probes on output shares of SecMux_n.

Share index	Glitch-extended probes on output shares
0	$x_0[i], \quad y_0[i], \quad z_0[i] = x_0[i] \oplus y_0[i]$
1	$x_1[i], \quad y_1[i], \quad z_1[i] = x_1[i] \oplus y_1[i]$

a crossing between share domains. We place the probe at the input of the corresponding register barrier, so that it glitch-extends through combinational logic according to the first cell in the last row of Table 6. We can then simulate this extended probe, as well as extended output probes on share 0 of each output (their extension is shown in Table 5), from the knowledge of inputs with share index 0, by sampling at random the values listed in the second cell of the same row: S_1, r_{j+n+1} for the chosen j, all $x_0[i]$ for $i \in [\![0, n-1]\!]$, and all $y_0[i]$ for $i \in [\![0, n-1]\!] \setminus \{j\}$. Sampling all of these quantities independently at random makes the simulation indistinguishable from the actual gadget execution, since in the latter case each value is blinded with fresh random bits. Finally, the simulator can compute $y_0[j]$ from its actual expression in Algorithm 5 since all terms of the expression have been simulated. Likewise, outputs $(z_0[i])_{0 \leqslant i < n}$ can be computed as $z_0 = x_0 \oplus y_0$.

The proof proceeds likewise for any other internal probe having even index, and is easy to adapt to probes having odd index by toggling the parity of all share indexes. □

Proposition 4. SecMux *is iterated glitch+transition-robust O-PINI.*

Proof. Follows from [CS21, Lemma 2] as SecMux is pipeline and glitch-robust O-PINI. □

3.3.3 Composition into secure modular addition

We show in Figure 3 the overall execution of the modular addition, based on the above defined gadgets. We call the composite gadget SecAdd_q. At the top of the figure are represented iterated executions of a single SecMux structural gadget, initially configured as a dual register (labeled as SDR) that holds the operands and results, and configured as a secure multiplexer (labeled as SMx) at the end of the algorithm, to perform the selection between the raw sum and the offsetted sum, in accordance with line 11 of Algorithm 3. The iterated executions of SDR are interconnected through a shifter, which implements the operation at lines 6 and 7 of Algorithm 3: shifting both operands right by one bit, and

Table 6: Simulation of a glitch-extended internal probe of even index in SecMux_n together with all extended probes on index-0 output shares. All input shares with index 0 ($(a_0[i])_{0\leqslant i<n}$, $(b_0[i])_{0\leqslant i<n}$, c_0, s_0) are known. The extended probes listed in the first column are placed at the input of the corresponding register barrier, and glitch-extend back to the inputs or to the previous register barrier.

Glitch-extended internal probes	Values simulated by random sampling	Notes
$S_0 = s_0 \oplus r_0$	$r_0, (x_0[i])_{0\leqslant i<n}$, $(y_0[i])_{0\leqslant i<n}$	Each $x_0[i]$, $y_0[i]$ is blinded with fresh randomness
$x_0[j] = a_0[j] \oplus b_0[j]S_0 \oplus r_{j+1}$ (j is fixed)	$r_0, r_{j+1}, (x_0[i])_{i\neq j}$, $(y_0[i])_{0\leqslant i<n}$	Compute $S_0 = s_0 \oplus r_0$, $x_0[j] = a_0[j] \oplus b_0[j] \cdot S_0 \oplus r_{j+1}$; each other $x_0[i]$, $y_0[i]$ is blinded with fresh randomness
$y_0[j] = b_0[j]S_1 \oplus r_{j+n+1}$ (j is fixed)	S_1, r_{j+n+1}, $(x_0[i])_{0\leqslant i<n}$, $(y_0[i])_{i\neq j}$	Compute $y_0[j] = b_0[j]S_1 \oplus r_{j+3}$; each other $x_0[i]$, $y_0[i]$ is blinded with fresh randomness; S_1 is blinded with r_0, which is not probed

setting their most significant bit to the sum bits s and δ output by SecDualFullAdder. Since this gadget is obviously share-isolating, it is also robustly O-PINI [CS21, Proposition 1]. The multiplexers that direct the flow of data between gadget executions are not represented: being robustly O-PINI by the same argument, their presence and position have no influence on the security of the composite gadget.

At the bottom, iterated executions of a single SecDualFullAdder structural gadget are shown: in the SDFA configuration, they perform the computation of the sums one bit at a time, in accordance with line 5 of Algorithm 3. Each execution gets its input carries c and d from the output carries z and ξ of the previous execution, except for their first execution, which gets 0 as input carries.

Due to the two-cycle latency from the a and b inputs of SecDualFullAdder to its s and δ outputs, the shift registers only need to store $n-2$ bits of each operand or result at the middle of the sum execution, which is done with the SecMux_{n-2} gadget. However, this situation is problematic at the beginning of the computation, while the pipeline of SecDualFullAdder is not full, and at its end, when the secure selection implementing modular reduction is performed. Both problems are solved by storing the extra bits inside the flip-flops already present in SecDualFullAdder. This explains the SDFAin configuration of the gadget at the first cycle, to store the most significant bit of each summand before the sum, and provide them one cycle later at its s and z outputs so they can be assigned to the most significant bit of the dual register. Likewise, toward the end of the addition, the SDFAcp configuration of SecDualFullAdder stores the least significant bit of the sums for two cycles, until the carry output of the offsetted sum is available.

When performing the modular reduction, the two extra bits must not only be stored, but the selection operation done by the secure multiplexer must also be performed on them. To do so, we use two properties of the design. First, the SecDualFullAdder gadget, which is no longer in use for ripple-carry addition at this stage, can be used to store one extra bit of each sum, and to perform the selection between them using logic that is compatible with the carry-computation logic. We denote this configuration by SDFAmx. Second, the two-cycle latency of the secure multiplexer (considered from the selection input to the result output) allows to fit two consecutive executions of SDFAmx in the same timespan, thereby doing the selection for the two missing bits of the sum.

Several aspects of the composite gadget have not been represented: in addition to the already mentioned multiplexers, whose presence is implied by the difference in wiring from cycle to cycle, some inputs and outputs of the gadgets have been omitted when they hold no relevant data. Since all gadgets are robustly O-PINI, how these omitted inputs and outputs are wired has no influence on the security of the composition. Finally, the

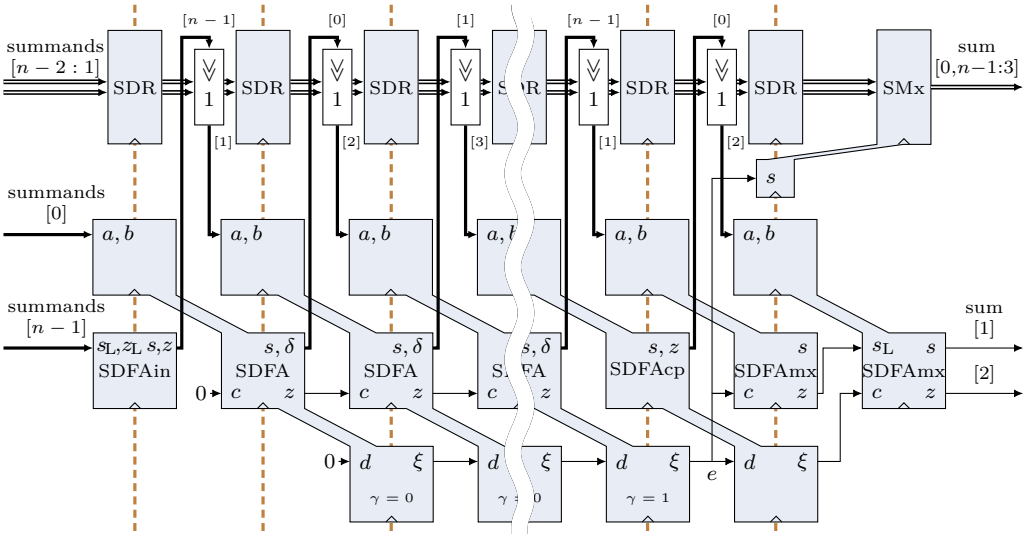

Figure 3: Gadget execution of secure modular addition (SecAdd_q). SDR and SMx are two configurations of the same SecMux_{n-2} gadget. Thin wires carry a sharing of a single bit; thick wires, a sharing of two bits; double wires, a sharing of $n-2$ bits. Numbers in brackets indicate the binary weight of the values carried on a wire. All gadgets are O-PINI, white ones being share-isolating. Parts of some gadget executions are not represented due to their output values being discarded.

logic supplying the prime modulus one bit at a time to SecDualFullAdder and governing the configuration of gadgets (choosing, for instance, between the dual-register and the secure-multiplexer functions of SecMux) is not shown: since this logic only processes public parameters, it has no impact on security. This leads us to our main result, Theorem 1.

Theorem 1. *Structural gadget* SecAdd_q *is glitch+transition-robust O-PINI.*

Proof. SecAdd_q is a structural gadget composition (its composing structural gadgets share no structural gates or wires). Since all its composing structural gadgets are iterated glitch+transition-robust O-PINI (by Proposition 2, Proposition 4, and the share-isolating characteristic of the other gadgets), the result follows from [CS21, Corollary 1]. □

3.4 Other secure summing operations

The SecAdd_q operation described in Subsubsection 3.3.3 can be adapted into secure modular subtraction SecSub_q with small adjustments that can be enabled or disabled at runtime. Comparing Equation 3 with Equation 1, the following changes can be listed: the computation of the raw result must be performed through subtraction $(a-b)$ instead of addition $(a+b)$; the computation of the offsetted result must add q instead of subtracting it; and, the selection between the raw and offsetted results must depend on the carry output by the raw subtraction, instead of the exclusive-or between the two output carries.

These modifications are implemented in the following way: by complementing the b input of SecDualFullAdder when in SDFA configuration (which is achieved by complementing one share of the value) and inputting a nonzero carry at the beginning of the sum execution, subtraction is computed instead of addition. Then, the last execution of SDFA is modified so that it copies the complement of its z output into its ξ output, which is achieved by configuring function $G^{m,\gamma=1}$ as $G^{m,1}(T_*, d_*, z_*) = (\overline{z_0}, z_1, 0, 0)$ and keeping $G^{m,\gamma=0}$ unchanged with respect to Table 2: this implements the modular-reduction condition in

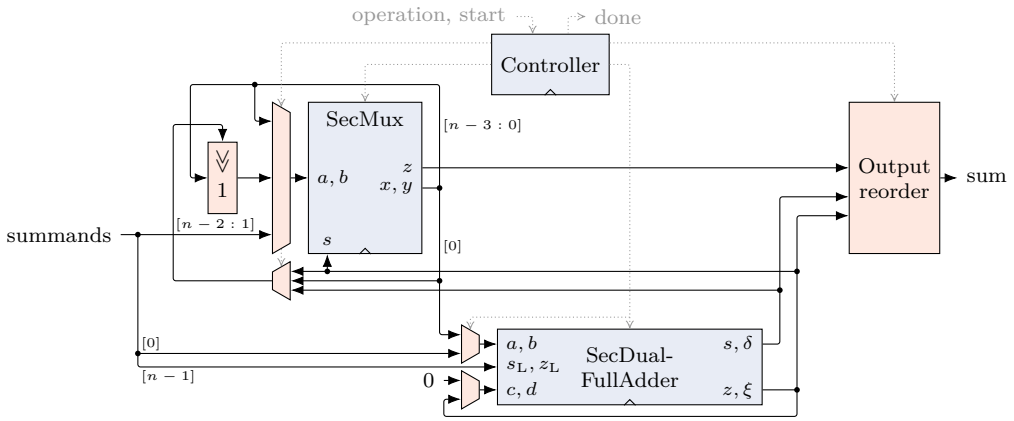

Figure 4: Simplified architecture diagram of our secure modular adder over Boolean shares. Light-blue blocks with a bottom notch are sequential components, while orange blocks are combinational only. Dashed lines carry control signals. Random bits are not represented.

accordance with Equation 3. Finally, instead of iterating over the bits of $2^n - q$ for the computation of the offsetted sum, the bits of q are used as source for the m parameter.

Furthermore, addition or subtraction can be computed modulo 2^n (these operations are named SecAdd and SecSub respectively) by setting $q = 0$ (i.e., by using parameter $m = 0$ for all executions of SecDualFullAdder).

The latency of power-of-two operations may be reduced by three cycles by stopping the execution of the algorithm as soon as the raw sum is available, and similarly, one cycle of latency can be saved when doing modular subtraction instead of modular addition, since the output carry of the raw sum is available one cycle earlier than that of the offsetted sum. In both cases, the position and ordering of the result bits within the dual shift register and SecDualFullAdder would be changed. We do not describe this solution in detail here.

The configurability among operations SecAdd_q, SecSub_q, SecAdd and SecSub is available at runtime for a very low area overhead: 8.6 % with the above-mentioned timing optimization, 0.4 % without[2]. As soon as the choice of operation is publicly known, this runtime configurability has no security implications since it is achieved with share-isolating gadgets (multiplexers, clearing or complementing of shares) inserted between the previously described O-PINI gadgets. Our design thus includes all four operations natively.

A simplified architectural diagram of our configurable secure adder is show in Figure 4 to clarify the interconnections between the gadgets. A controller, that includes a cycle counter and all the logic to generate the control signals (including the enumeration of the bits of the modulus), chooses the configuration of the two gadgets at each cycle. It also drives the multiplexers directing the flow of data between gadgets, and choosing the correct ordering of the result bits for the requested operation. Disabling the timing optimizations mentioned above removes the output-reordering unit, since in this case, all operations have the same latency and the alignment of the result within the shift register is always the same.

3.5 Boolean-to-Arithmetic and Arithmetic-to-Boolean conversions

One core application of secure addition over Boolean shares is the implementation of secure conversion between masking schemes, namely Boolean-to-Arithmetic (B2A) and Arithmetic-to-Boolean (A2B) conversions. These conversion operations were first described by Goubin [Gou01] in the restricted case of first-order masking. Later on, Coron *et al.* [CGV14] showed a new method, applicable to any masking order, that relied on secure addition over

[2]Total area overhead for an ASIC implementation.

Algorithm 6: Mod-q-Arithmetic-to-Boolean conversion (A2B$_q$) using SecAdd$_q$

Input: $a_* \in [\![0, q-1]\!]^2$
Output: $b_* \in \mathcal{B}_2([\![0, q-1]\!])$ with $b_0 \oplus b_1 = a_0 + a_1 \bmod q$

1 $m_0 \leftarrow \mathbb{F}_2^n \qquad m_1 \leftarrow \mathbb{F}_2^n$
2 $\text{augend}_* = a_0 \oplus m_0, m_0$
3 $\text{addend}_* = a_1 \oplus m_1, m_1$
4 $b_0, b_1 = \text{SecAdd}_q(\text{augend}_*, \text{addend}_*)$
// At this point $b_0 \oplus b_1 = a$
5 **return** b_0, b_1

Algorithm 7: Boolean-to-Mod-q-Arithmetic conversion (B2A$_q$) using SecSub$_q$

Input: $b_* \in \mathcal{B}_2([\![0, q-1]\!])$
Output: $a_* \in [\![0, q-1]\!]^2$ with $a_0 + a_1 \bmod q = b_0 \oplus b_1$

1 $r \leftarrow [\![0, q-1]\!] \qquad m \leftarrow \mathbb{F}_2^n$
2 $\text{addend}_* = r \oplus m, m$
3 $s_0, s_1 = \text{SecSub}_q(b_*, \text{addend}_*)$
// At this point $s_0 \oplus s_1 = (b - r) \bmod q$
4 $a_0, a_1 = s_0 \oplus s_1, r$
5 **return** a_0, a_1

Boolean shares. This solution was extended to modular arithmetic masking by Barthe *et al.* [BBE+18] by using secure modular addition SecAdd$_q$ instead of secure power-of-two addition: the former being implemented with two consecutive executions of the latter.

We recall in algorithms 6 and 7 the implementation of A2B$_q$ and B2A$_q$ for first-order security [FBR+22]. In the latter case, we slightly simplified the algorithm by using secure modular subtraction SecSub$_q$ instead of negation modulo q and secure modular addition.

The corresponding conversions are carried out similarly in the case of power-of-two arithmetic masking, by using SecAdd (respectively SecSub) instead of SecAdd$_q$ (respectively SecSub$_q$) and sampling mask r from $\mathbb{F}_2^n$. Our circuit, that provides the four operations SecAdd, SecSub, SecAdd$_q$ and SecSub$_q$, thus supports all four of these masking conversions.

4 Comparison with previous works

In this section, we compare the performance of our design with the literature. As we are not aware of any previous work that directly implements secure modular addition, we start our comparison in the context of power-of-two addition, and extend it to modular addition by giving the actual performance of our work, and the estimated performance of previous works in this context based on generic conversions from power-of-two to modular addition. We report synthesis results both for an ASIC in a 40 nm technology (the primary target), and for FPGA targets of the Artix-7 family (XC7A100T, speed grade -3) to ease the comparison with previous works[3]. For area comparisons, we mainly focus on the number of flip-flops used. Absent any better way to compare areas between different ASIC and FPGA technologies, we consider this measurement a decent indicator of the overall area[4]. For more accurate comparisons, we also give the ASIC areas in the gate-equivalent (GE) unit, and the LUT counts of our FPGA implementations.

During synthesis, we took special precautions to prevent the synthesizer from optimizing logic equations in ways that introduce vulnerabilities not present in the register-transfer level description of the design. Practically, we isolated into a separate submodule the logic equation defining each output share of nonlinear gadgets. Then, for the ASIC synthesis, we selectively disabled logic optimizations across these module boundaries, while still allowing other optimizations that do not compromise security. For the FPGA synthesis, we did not find an equivalent synthesis option and had to disable all cross-boundary optimization.

Since our design is meant to be integrated into a larger circuit containing other secure components, we assume a random number generator to be already present, and do not take into account the additional hardware area needed for it.

[3] Spartan-6 and Artix-7 FPGAs have similarly-capable lookup tables: LUT counts should be comparable.
[4] The flip-flops take up 40 % of the overall area of our ASIC implementations.

Table 7: ASIC Performance of first-order–secure 32-bit addition over Boolean sharings

Design	Technology	Flip-flops	Area kGE	Latency cycles	Random bit/cyc.	Freq. MHz	Notes
Ripple carry adder							
Ours	40 nm CMOS	146	2.05	33	69	400	Mod-q add.: 36 cycles
Ours (no opt)	40 nm CMOS	146	1.90	36	69	400	No timing optimization
[CGM+23]	Nangate 45	3100**	19.23	31*	32		Fully pipelined
Fully pipelined carry-lookahead adders							
[CGM+23]	Nangate 45		18.30	5*	374		Kogge-Stone
[CGM+23]	Nangate 45		13.77	6*	172		Sklansky
[CGM+23]	Nangate 45		12.07	9*	115		Brent-Kung

* Throughput of one addition per cycle. ** Figure absent from paper, estimated from the description.

4.1 Power-of-two addition

Thanks to the area efficiency of the ripple-carry adder architecture, we obtain a very small hardware design that performs n-bit addition with a latency of $n+1$ clock cycles. We give in Table 7 and Table 8 the area utilization and latency we obtain for ASIC and FPGA implementations respectively, compared with previous works. Our figures are for 32-bit addition to match the literature; however, our work is better suited to the smaller integer widths encountered in lattice-based cryptography, e.g. 12 or 23 bits [SAB+20, LDK+20]. When it includes the timing optimizations for subtraction and power-of-two operations, mentioned in Subsection 3.4, the ASIC design takes up 2.05 kGE and executes power-of-two operations in 33 cycles, and modular subtraction and addition in 35 and 36 cycles respectively. Without these optimizations, all operations have a latency of 36 cycles, but the design only occupies 1.90 kGE due to having less combinational logic.

The above ASIC synthesis results are reported for a target frequency of 400 MHz in the worst PVT corner; however, adjusting the synthesis constraints allows to reach up to 660 MHz at the cost of increasing the area by 50 % (or by 40 % for the design without timing optimizations), still in the worst corner.

As expected, our ripple-carry adder has six to thirty times smaller area than previous works based on pipelined parallel-prefix adders [SMG15, FBR+22, BG22, CGM+23], at the expected cost of much higher latency. We also reduce the number of flip-flops by one third with respect to the ripple-carry adder of Schneider *et al.* [SMG15], thanks to the use of two shares per secret value instead of three for their threshold implementation. This lower number of shares still achieves the same security order, and additionally benefits from a proof of robustness against transitions and glitches, which is not explicitly the case for [SMG15]. Indeed, threshold implementations are not robustly composable, and while Schneider *et al.* discuss how they avoid transition-based leakage in a specific part of their design, they do not provide a full robustness analysis[5]. With and without timing optimization, our work uses 1.95 and 1.67 times as many LUTs as that of [SMG15], but it is difficult to know how much additional logic it represents since they do not synthesize for ASIC. Anyhow, this extra logic area is a low price to pay for the additional modular reduction.

Since the other designs from the state of the art are fully pipelined, they can perform one addition per clock cycle in steady operation, compared to one addition per 33 or 32 clock cycles for our iterative ripple-carry adder and that of Schneider *et al.*. However, such architectures are only possible for the highest-performance applications considering the

[5] Replacing the TI gadgets used by Schneider *et al.* with glitch-robust O-PINI gadgets to get provable robustness against glitches and transitions is not directly possible since glitch-robust O-PINI gadgets from the literature [CS21, KM22] have two or three cycles of latency, while the construction of [SMG15] requires the gadgets in the carry-computation path to have a single cycle of latency.

Table 8: FPGA Performance of first-order–secure 32-bit addition over Boolean sharings

Design	Family	Area		Latency	Random	Freq.	Notes
		Flip-flops	LUTs	cycles	bit/cyc.	MHz	
Ripple carry adder							
Ours	Artix-7	146	441	33	69	200	Mod-q addition: 36 cycles
Ours (no opt)	Artix-7	146	380	36	69	250	No timing optimization
[SMG15]	Spartan-6	223	227	32	4	101	Refreshes only at first cycle
Fully pipelined Kogge-Stone carry-lookahead adder							
[SMG15]	Spartan-6	1330	937	6*	31	62	
[FBR+22]	Artix-7	1323	2464	6*		454	Contains additional logic
TI [BG22]	Spartan-6	1416	873	6*	32	228	
HPC2 [BG22]	Spartan-6	3981	2936	12*	249	176	
Fully pipelined Sklansky carry-lookahead adder							
TI [BG22]	Spartan-6	1416	579	6*	41	174	
HPC2 [BG22]	Spartan-6	3166	1801	12*	119	153	
Fully pipelined Brent-Kung carry-lookahead adder							
TI [BG22]	Spartan-6	2352	487	9*	31	280	
HPC2 [BG22]	Spartan-6	4317	1588	18*	74	173	

* Throughput of one addition per cycle.

very large area occupied by fully pipelined adders. Our work, instead, definitely achieves its primary goal of low area utilization.

In terms of randomness usage, our proposed construction requires 8 fresh random bits per cycle for SecDualFullAdder and $2n-3$ for SecMux_{n-2}, that is, $2n+5$ bits per cycle in total. At 69 bits per cycle, the randomness requirements of our construction are consistent with the other listed designs, that range from 4 to 374 bits per cycle[6]. These relatively high randomness requirements are due to using an iterative design, since it mandates using O-PINI gadgets and sizing the amount of refresh bits based on the most costly iteration.[7]

4.2 Modular addition

As its primary objective, our architecture allows for performing secure modular addition at nearly no extra cost with respect to power-of-two addition: the necessary area is already included, and the latency is increased by three cycles only. This is in contrast to previous works, which do not directly support modular addition. They must instead rely on two parallel or consecutive power-of-two additions, which uses either double the area (and double the per-cycle randomness), or double the latency with half the throughput. Consequently, in this setting, our construction achieves an even closer cost/performance ratio to high-performance designs from the literature: our design uses 5256 flip-flop $\times$ cycles per modular addition, and the efficient construction of [FBR+22] would spend at least 2646 flip-flop $\times$ cycles (in steady state) for the same task. Our work thus compares unexpectedly well with high-performance designs considering its focus on area minimization.

A similar comparison can be made with the ripple-carry adder of Schneider *et al.* [SMG15]. In this case, we will study two constructions for modular addition: either the

[6] While the cumulative randomness requirements per addition of our solution is very high (2277 bits per 32-bit addition), we consider this figure of little importance: without expensive buffering, the random number generator must provide the required per-cycle randomness, and cumulative randomness is irrelevant.

[7] Analysing the area required for randomness generation is beyond our scope; however, extrapolating the research of Cassiers *et al.* [CMM+23] suggests that around 1.6 kGE would be required for a 138-bit linear-feedback shift register providing 69 bits per cycle, or around 4 kGE for an unrolled Trivium cipher.

raw and offsetted sums are computed in parallel, and secure selection between these two results accomplishes the modular reduction (Subsection 3.1); or, using the solution of [FBR+22], the offsetted sum is computed directly, and this result is conditionally added with the modulus. The first solution will require the side-by-side instantiation of two adders and a secure n-bit multiplexer, and add at least one cycle of latency to the whole operation. The second one, instead, may reuse the same secure adder for both additions, but it will double the latency and halve the throughput with respect to power-of-two addition. Both solutions will thus use more than 14 272 flip-flop × cycles per 32-bit modular addition, which represents nearly three times the overall cost of our proposal.

5 Leakage assessment

We have proved in Subsubsection 3.3.3 the security of our construction in a strong security model that is robust against glitches and transitions. However, since hardware implementations may have additional defects overlooked by this model, we give further assurance in the security of our design by performing a leakage assessment in simulation. Since this gate-level simulation is noiseless and it models the propagation delays within the cells for each input condition (not the routing delays, as simulation is before place-and-route), we expect better fidelity with this approach than by porting the design to an FPGA for validation.

We thus synthesize the secure adder for $n = 12$ bits with prime modulus $q = 3329$, for a 40 nm CMOS technology, and annotate it with gate-timing information. We then simulate the obtained netlist and derive power-consumption traces from the toggle count of the circuit, that is, the number of nets that change logic value at each time sample. This so-called *toggle-count* metric, introduced by Sadhukhan *et al.* [SMRM19], is suitable for the leakage assessment of a pre-silicon design. The simulation assumes a clock frequency of 200 MHz, easily achieved by the synthesized design, and uses a time resolution of 1 ps.

We analyze the simulated power traces in the test-vector leakage assessment (TVLA) methodology [GJJR11], which consists in collecting two separate sets of traces for different scenarios, and performing a t-test between the two sets to check whether they are statistically distinguishable. In all our experiments, the first set of traces corresponds to summing two all-zero operands, each masked with a uniformly random Boolean mask; the second set of traces corresponds to sampling the two summands independently and uniformly at random from $[\![0, q-1]\!]$, again masking each of them with a random Boolean mask.

The results of the TVLA are reproduced in Figure 5. On the left are shown the t-test results with sets of 250 million traces each, at the first order. Since the power trace contains a large number of samples (85 000, of which 25 000 are nonzero), we choose a threshold of 4.75 for the t statistic, which corresponds to a false-positive probability of 5 % [DZD+18]. As expected from a secure design, the t statistic does not cross the ± 4.75 threshold anywhere in the trace (its maximum absolute value is 4.02), showing no statistically significant difference in the average power consumption between the two sets of traces.

To make the leakage detection more sensitive, we follow the more advanced methodology of [DZD+18] and compute the Higher Criticism (HC) statistic. Instead of only considering the extreme values of the t-test, this methodology checks the distribution of all t values to ensure that they support the joint null hypothesis of having no leakage at any trace point[8]. This statistic is equal to 1.6 in the first-order t-test, which is well below the 4.8 threshold for a significance level of 5 %. Furthermore, our choice of a false-positive probability of 5 % is more conservative (*i.e.* more sensitive to leakage) than the 1 % chosen by [DZD+18], which would result in threshold values of 5.1 for the t statistic and 10.1 for the HC statistic.

[8] The HC statistic implicitly assumes that all points of the trace have independently distributed noise; however, it is not significantly affected by local correlation in the noise [DZD+18].

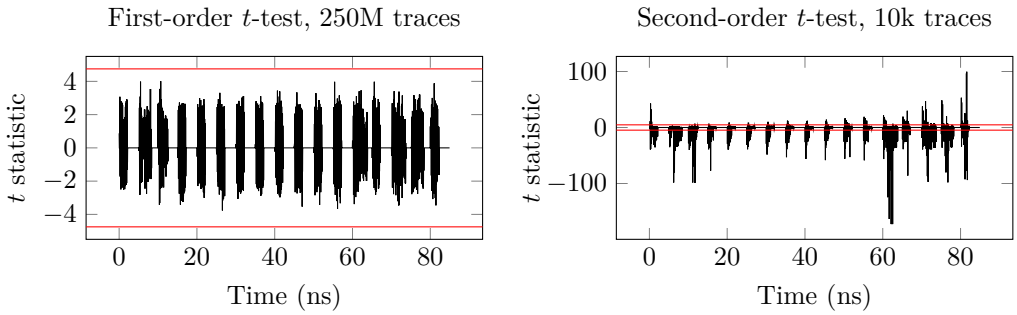

Figure 5: Leakage-assessment results of secure modular addition with first-order and second-order t-test at 200 MHz with $n = 12$. Modulus is $q = 3329$. A fixed-vs-random scenario is studied: for the first set of traces, both summands are always zero; for the second set, both summands are uniformly and independently sampled from $[\![0, q-1]\!]$. As expected for a secure first-order design, the first-order t-test shows no leakage ($|t| \leqslant 4.02 < 4.75$) with 250 million traces, while the second-order one already exhibits strong leakage ($\max |t| \gg 4.75$) at ten thousand traces. The HC statistic is respectively $1.6 < 4.8$ and ∞.

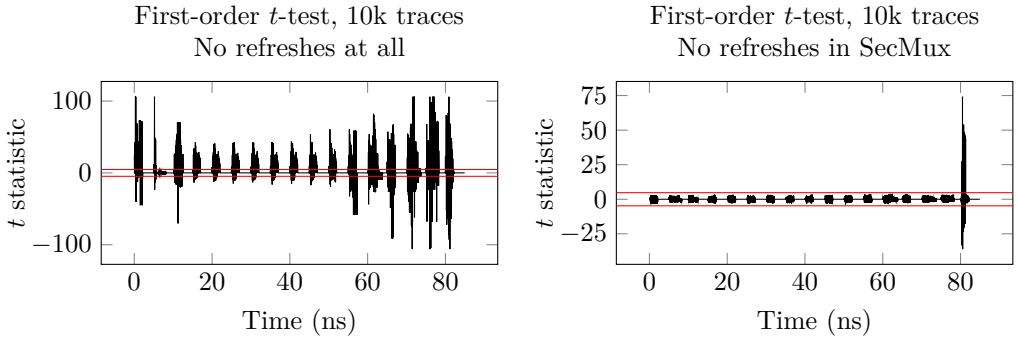

Figure 6: Leakage-assessment results of secure modular addition at the first order, with refreshes totally or partially disabled. Zero-vs-random scenario with ten thousand traces per set. With all refreshing disabled, leakage occurs at every clock cycle (left), while solely disabling the refreshing of SecMux contains the leakage to the last cycle (right).

On the right of Figure 5, a second-order TVLA is conducted: as anticipated, strong leakage is shown since our design is only secure at the first order. In this situation, the small number of ten thousand traces per set is amply sufficient to detect the leakage, with t values exhibiting several strong peaks whose amplitude exceeds ± 100, well beyond the threshold for leakage detection. This is again confirmed with absolute certainty by the HC statistic, which is too large to even be represented as a floating-point value.

We run two additional experiments in Figure 6, by totally or selectively deactivating the random bits for refreshes. Both experiments involve ten thousand traces per set. In the left graph, we show the effect of deactivating all refreshes in the masked circuit, with the expected outcome of introducing strong leakage, since the security of the nonlinear masked operations in this work entirely depends on proper refreshing of the shares. In the rightmost graph, instead, we only deactivate part of the refreshes: specifically, the $2n - 3$ refresh bits used by the SecMux gadget, while the refreshes inside SecDualFullAdder are kept. This partial disabling still introduces strong leakage, but this time, restricted to the last two clock cycles of secure modular addition. These are the cycles in which modular reduction is performed by selecting between the raw and offsetted sums. Since it is the only time at which a refreshing is required for security within SecMux, the leakage only

occurs at these two cycles[9]. If the design were restricted to power-of-two operations, the randomness requirements would thus drop to 8 fresh bits per cycle without loss of security.

Overall, this leakage assessment helps confirm that the glitch+transition-robust probing model does not overlook glaring hardware defects, and that the synthesis flow does not introduce obvious vulnerabilities not present in the register-transfer level design.

6 Conclusions

6.1 Summary

In this work, we have presented a new construction to compute modular addition securely over Boolean shares, with proven security against first-order probing attacks in the robust-probing model. To the best of our knowledge, this is both the first secure adder natively supporting modular arithmetic, and the first iterative adder benefiting from a proof of robust security in the presence of glitches and transitions. We furthermore demonstrated that these security claims firmly hold when performing leakage assessment in simulation.

Through careful design, our construction reaches an area efficiency that is significantly beyond the state of the art without compromising on security, and with better flexibility than any of the previous works from the literature, as it natively supports runtime configurability among either addition or subtraction, and either reduction modulo a publicly-known prime, or simple wraparound modulo a power of two.

Secure modular addition and subtraction, while being costly operations, are crucial to the protection of recent lattice-based cryptography algorithms. We expect that our work will help implement these algorithms securely on resource-constrained embedded devices.

6.2 Open problems

This study exclusively focuses on first-order security, which was deemed sufficient in the context of a low-cost implementation. An important extension of this work would consist in determining how the architecture of the solution would change when generalizing it to higher security orders. Indeed, our performance constraints forced us to design custom integrated gadgets, whose applicability to higher-order masking is unclear. The main step toward this goal would be to generalize SecDualFullAdder to high-order masking, while still containing to a single cycle the latency of its carry-chain paths $c \to z$ and $d \to \xi$.

We have shown in Section 4 that, while our solution provides less throughput per unit of area than carry-lookahead adders from the literature, this difference remains extremely reasonable given the very large starting area of these high-performance adders. It would thus be interesting to explore whether intermediate approaches could lead to better efficiency than either architecture in moderate-performance applications. Besides, a strong assumption that was made in our work is that fresh random bits have low cost due to an already present pseudorandom number generator. For contexts in which this assumption does not hold, it would be highly beneficial to reduce the randomness requirements of our individual gadgets, or to reuse randomness between gadgets.

Finally, our main concern in this work was to reduce the area consumption of secure modular addition; yet, an equally important concern for low-cost implementations is their power consumption. Considering to the shift-register structure of our construction, about half of the flip-flops are expected to toggle at each clock cycle, which may cause a high dynamic power consumption compared to the small size of the circuit. Whether this consumption can be tolerated will be highly application-dependent.

[9]Since the secure dual register is also implemented by the SecMux gadget in its SDR configuration, refreshing also occurs in this other configuration, but is not required for security. Indeed, if its SMx configuration is removed, SecMux becomes share-isolating, so it remains O-PINI even without refreshes.

Acknowledgements

The authors would like to thank Ruggero Susella, Nicolas Bruneau and Guillaume Reymond, among others, for their ideas and support. We are also grateful to the anonymous reviewers for their thorough work and the kind advice they provided. This work has been partially supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02).

A Explicit definition of SecDualFullAdder configurations

Algorithm 8: SDFA function of SecDualFullAdder

Input: Shares $a_*, b_*, c_*, d_* \in \mathcal{B}_2(\mathbb{F}_2)$, public param. $m, \gamma \in \mathbb{F}_2$, randomness $r_0, \ldots, r_7 \in \mathbb{F}_2$
Output: Shares $s_*, z_*, \delta_*, \xi_* \in \mathcal{B}_2(\mathbb{F}_2)$ such that $z \parallel s = a + b + c$ and $\xi \parallel (\delta \oplus s) = ((a \oplus b \oplus c) + d + m) \oplus (\gamma \cdot z \ll 1)$

$A_* = \mathrm{Reg}[\mathrm{Refresh}(a_*, r_0)] \qquad B_* = \mathrm{Reg}[\mathrm{Refresh}(b_*, r_1)]$
$S_* = A_* \oplus B_* \oplus c_*$
$Z_0, Z_1 = A_0 \cdot B_0 \oplus A_0 \cdot c_0 \oplus B_0 \cdot c_0, A_1 \cdot B_1 \oplus A_1 \cdot c_1 \oplus B_1 \cdot c_1$
$Z_2, Z_3 = A_1 \cdot (B_0 \oplus c_0), A_0 \cdot (B_1 \oplus c_1)$
$Z_4, Z_5 = B_1 \cdot c_0, B_0 \cdot c_1$
$T_* = \mathrm{Reg}[A_* \oplus B_* \oplus c_*]$
$s_* = \mathrm{Reg}[\mathrm{Refresh}(S_*, r_7)]$
$z_* = \mathrm{Reg}[\mathrm{Refresh}((Z_0, Z_1), r_2)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_2, Z_3), r_3)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_4, Z_5), r_4)]$
$\delta_* = d_* \oplus (m, 0)$
$\Xi_0, \Xi_1 = (T_0 \oplus d_0) \cdot m \oplus T_0 \cdot d_0 \oplus \gamma \cdot z_0, (T_1 \oplus d_1) \cdot m \oplus T_1 \cdot d_1 \oplus \gamma \cdot z_1$
$\Xi_2, \Xi_3 = T_1 \cdot d_0, T_0 \cdot d_1$
$\xi_* = \mathrm{Reg}[\mathrm{Refresh}((\Xi_0, \Xi_1), r_5)] \oplus \mathrm{Reg}[\mathrm{Refresh}((\Xi_2, \Xi_3), r_6)]$
return $s_*, z_*, \delta_*, \xi_*$

Algorithm 9: SDFAmx function of SecDualFullAdder

Input: Shares $a_*, b_*, c_*, s_{\mathrm{L},*} \in \mathcal{B}_2(\mathbb{F}_2)$, randomness $r_0, \ldots, r_7 \in \mathbb{F}_2$
Output: Shares $s_*, z_* \in \mathcal{B}_2(\mathbb{F}_2)$ such that $z = a \cdot \overline{s} \oplus b \cdot s$ and $s = s_{\mathrm{L}}$

$A_* = \mathrm{Reg}[\mathrm{Refresh}(a_*, r_0)] \qquad B_* = \mathrm{Reg}[\mathrm{Refresh}(b_*, r_1)]$
$S_* = s_{\mathrm{L}}$
$Z_0, Z_1 = A_0 \cdot \overline{c_0} \oplus B_0 \cdot c_0, A_1 \cdot c_1 \oplus B_1 \cdot c_1$
$Z_2, Z_3 = A_1 \cdot \overline{c_0}, A_0 \cdot c_1$
$Z_4, Z_5 = B_1 \cdot c_0, B_0 \cdot c_1$
$s_* = \mathrm{Reg}[\mathrm{Refresh}(S_*, r_7)]$
$z_* = \mathrm{Reg}[\mathrm{Refresh}((Z_0, Z_1), r_2)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_2, Z_3), r_3)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_4, Z_5), r_4)]$
return s_*, z_*

Algorithm 10: SDFAcp function of SecDualFullAdder

Input: Shares $a_*, b_*, c_*, d_* \in \mathcal{B}_2(\mathbb{F}_2)$, randomness $r_0, \ldots, r_7 \in \mathbb{F}_2$
Output: Shares $s_*, z_*, \xi_* \in \mathcal{B}_2(\mathbb{F}_2)$ such that $s = a$, $z = b$, $\xi = d$

$A_* = \mathrm{Reg}[\mathrm{Refresh}(a_*, r_0)] \qquad B_* = \mathrm{Reg}[\mathrm{Refresh}(b_*, r_1)]$
$S_* = A_*$
$Z_0, \ldots, Z_5 = B_0, B_1, 0, \ldots, 0$
$s_* = \mathrm{Reg}[\mathrm{Refresh}(S_*, r_7)]$
$z_* = \mathrm{Reg}[\mathrm{Refresh}((Z_0, Z_1), r_2)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_2, Z_3), r_3)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_4, Z_5), r_4)]$
$\Xi_0, \ldots, \Xi_3 = d_0, d_1, 0, 0$
$\xi_* = \mathrm{Reg}[\mathrm{Refresh}((\Xi_0, \Xi_1), r_5)] \oplus \mathrm{Reg}[\mathrm{Refresh}((\Xi_2, \Xi_3), r_6)]$
return s_*, z_*, ξ_*

Algorithm 11: SDFAin function of SecDualFullAdder

Input: Shares $s_{\mathrm{L},*}, z_{\mathrm{L},*} \in \mathcal{B}_2(\mathbb{F}_2)$, randomness $r_0, \ldots, r_7 \in \mathbb{F}_2$
Output: Shares $s_*, z_* \in \mathcal{B}_2(\mathbb{F}_2)$ such that $s = s_{\mathrm{L}}$, $z = z_{\mathrm{L}}$

$S_* = s_{\mathrm{L},*}$
$Z_0, \ldots, Z_5 = z_{\mathrm{L},0}, z_{\mathrm{L},1}, 0, \ldots, 0$
$s_* = \mathrm{Reg}[\mathrm{Refresh}(S_*, r_7)]$
$z_* = \mathrm{Reg}[\mathrm{Refresh}((Z_0, Z_1), r_2)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_2, Z_3), r_3)] \oplus \mathrm{Reg}[\mathrm{Refresh}((Z_4, Z_5), r_4)]$
return s_*, z_*

References

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_12.

[Bed62] O. J. Bedrij. Carry-select adder. *IRE Transactions on Electronic Computers*, EC-11(3):340–346, 1962. doi:10.1109/IRETELC.1962.5407919.

[BG22] Florian Bache and Tim Güneysu. Boolean masking for arithmetic additions at arbitrary order in hardware. *Applied Sciences*, 12(5), 2022. URL: https://www.mdpi.com/2076-3417/12/5/2274, doi:10.3390/app12052274.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):173–214, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064. doi:10.46586/tches.v2021.i4.173-214.

[BMRT22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. IronMask: Versatile verification of masking security. In *2022 IEEE Symposium on Security and Privacy*, pages 142–160. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833600.

[CFOS21] Gaëtan Cassiers, Sebastian Faust, Maximilian Orlt, and François-Xavier Standaert. Towards tight random probing security. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 185–214, Virtual Event, August 2021. Springer, Heidelberg. doi:10.1007/978-3-030-84252-9_7.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Transactions on Computers*, 70(10):1677–1690, 2021. doi:10.1109/TC.2020.3022979.

[CGM+23] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Reducing area and latency of masked pipelined circuits. Cryptology ePrint Archive, Report 2023/1600, 2023. https://eprint.iacr.org/2023/1600.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and*

Embedded Systems – CHES 2014, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, Heidelberg, September 2014. doi:10.1007/978-3-662-44709-3_11.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1_26.

[CMM+23] 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. Cryptology ePrint Archive, Paper 2023/1134, 2023. URL: https://eprint.iacr.org/2023/1134.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020. doi:10.1109/TIFS.2020.2971153.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):136–158, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8790. doi:10.46586/tches.v2021.i2.136-158.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, Heidelberg, May 2014. doi:10.1007/978-3-642-55220-5_24.

[DZD+18] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In *Smart Card Research and Advanced Applications: 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13–15, 2017, Revised Selected Papers*, pages 105–122, Cham, 2018. Springer. doi:10.1007/978-3-319-75208-2_7.

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):414–460, 2022. doi:10.46586/tches.v2022.i1.414-460.

[FG05] Wieland Fischer and Berndt M. Gammel. Masking at gate level in the presence of glitches. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 187–200. Springer, Heidelberg, August / September 2005. doi:10.1007/11545262_14.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270. doi:10.13154/tches.v2018.i3.89-120.

[FIP23a] Module-lattice-based key-encapsulation mechanism. National Institute of Standards and Technology, draft of NIST FIPS PUB 203, U.S. Department of Commerce, August 2023.

[FIP23b] Module-lattice-based digital signature standard. National Institute of Standards and Technology, draft of NIST FIPS PUB 204, U.S. Department of Commerce, August 2023.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136. Cryptography Research Inc., 2011. URL: https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, Heidelberg, May 2001. doi:10.1007/3-540-44709-1_2.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_27.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1799–1812. ACM Press, November 2022. doi:10.1145/3548606.3559362.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, Heidelberg, December 2020. doi:10.1007/978-3-030-64837-4_26.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *ESORICS'98: 5th European Symposium on Research in Computer Security*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer, Heidelberg, September 1998. doi:10.1007/BFb0055858.

[LB61] M. Lehman and N. Burla. Skip techniques for high-speed carry-propagation in binary arithmetic units. *IRE Transactions on Electronic Computers*, EC-10(4):691–698, 1961. doi:10.1109/TEC.1961.5219274.

[LDK+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Mac61] O. L. MacSorley. High-speed arithmetic in binary computers. *Proceedings of the IRE*, 49(1):67–91, 1961. doi:10.1109/JRPROC.1961.287779.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_9.

[SAB+20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Skl60] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960. doi:10.1109/TEC.1960.5219822.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking – towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15: 13th International Conference on Applied Cryptography and Network Security*, volume 9092 of *Lecture Notes in Computer Science*, pages 559–578. Springer, Heidelberg, June 2015. doi:10.1007/978-3-319-28166-7_27.

[SMRM19] Rajat Sadhukhan, Paulson Mathew, Debapriya Basu Roy, and Debdeep Mukhopadhyay. Count your toggles: a new leakage model for pre-silicon power analysis of crypto designs. *Journal of Electronic Testing*, 35(5):605–619, 2019. doi:10.1007/s10836-019-05826-8.

[WT90] B.W.Y. Wei and C.D. Thompson. Area-time optimal adder design. *IEEE Transactions on Computers*, 39(5):666–675, 1990. doi:10.1109/12.53579.