



Efficient Boolean-to-Arithmetic Mask Conversion in Hardware

Aein Rezaei Shahmirzadi  and Michael Hutter 

PQShield, Oxford, UK

Abstract. Masking schemes are key in thwarting side-channel attacks due to their robust theoretical foundation. Transitioning from Boolean to arithmetic (B2A) masking is a necessary step in various cryptography schemes, including hash functions, ARX-based ciphers, and lattice-based cryptography. While there exists a significant body of research focusing on B2A software implementations, studies pertaining to hardware implementations are quite limited, with the majority dedicated solely to creating efficient Boolean masked adders. In this paper, we present first- and second-order secure hardware implementations to perform B2A mask conversion efficiently without using masked adder structures. We first introduce a first-order secure low-latency gadget that executes a $B2A_{2^k}$ in a single cycle. Furthermore, we propose a second-order secure $B2A_{2^k}$ gadget that has a latency of only 4 clock cycles. Both gadgets are independent of the input word size k . We then show how these new primitives lead to improved $B2A_q$ hardware implementations that perform a B2A mask conversion of integers modulo an arbitrary number. Our results show that our new gadgets outperform comparable solutions by more than a magnitude in terms of resource requirements and are at least 3 times faster in terms of latency and throughput. All gadgets have been formally verified and proven secure in the glitch-robust PINI security model. We additionally confirm the security of our gadgets on an FPGA platform using practical TVLA tests.

Keywords: Secure Mask Conversion · Mixed Boolean Arithmetic (MBA) · Boolean-to-Arithmetic (B2A) · Arithmetic-to-Boolean (A2B) · Side-Channel Analysis · DPA Countermeasures · Hardware Implementation · Physical Security

1 Introduction

Side-channel attacks, which exploit physical information leakage like power consumption or electromagnetic radiation from cryptographic implementations, pose a significant security threat. Common types of side-channel attacks include timing attacks [Koc96], which exploit variations in execution time, EM, or power analysis attacks [KJJ99].

The standard countermeasure against these attacks is *masking*, where secret-shared intermediate variables are computed, ensuring they remain independent of the secret data manipulated by the device. The application of masking in cryptographic operations varies based on the type of operations performed. Logical operations like XOR and shifts are safeguarded using Boolean masking, while additions, subtractions, and multiplications require arithmetic and multiplicative masking, respectively. However, cryptographic algorithms often involve a combination of these operations, necessitating the conversion of masks between Boolean and arithmetic forms to get correct results. Typical examples are symmetric primitives such as SHA-2, Blake [AHMP10], Skein [FLS⁺10], XTEA [NW97], or ChaCha20 [Ber08], which all use a moduli with a power-of-two. Other examples are

E-mail: aein.shahmirzadi@pqshield.com (Aein Rezaei Shahmirzadi), michael.hutter@pqshield.com (Michael Hutter)



lattice-based cryptographic schemes in which polynomial multiplication and addition are preferably performed on arithmetic shares, while binomial sampling, for example, requires Boolean masking. These schemes often use a modulus different than a power-of-two and require additional modulo reductions. Consequently, there is a demand for efficient techniques that facilitate the conversion between arithmetic and Boolean sharing types in both fields of characteristic two and prime fields.

In 2001, Goubin [Gou01] introduced a highly efficient first-order secure $B2A_{2^k}$ mask conversion method. Notably, this solution, which remains independent of the input word size k , has a runtime complexity of $\mathcal{O}(1)$. Additionally, Goubin proposed a first-order secure $A2B_{2^k}$ solution that used a masked Ripple-Carry Adder (RCA). Due to the carry chain, the runtime complexity depends on k and is therefore $\mathcal{O}(k)$. This was further improved by Coron et al. [CGTV15] who suggested to employ a masked carry-lookahead adder in 2015, known as the **SecAdd** gadget, achieving a lower logarithmic complexity of $\mathcal{O}(\log k)$. This specialized gadget facilitates both additions and subtractions on Boolean masked shares and can be used to perform both $B2A_{2^k}$ and $A2B_{2^k}$ mask conversions. Several works, including [BDCU17, WH17, KSM19], analyzed different types of masked adders and proposed additional improvements and variants of the **SecAdd** gadget to enhance implementation efficiency.

The exploration of mask conversion algorithms for numbers modulo a prime p was initiated by Barthe et al. [BBE⁺18]. They introduced a variant of the **SecAdd** gadget capable of also performing modulo reduction, which is needed to mask many cryptographic schemes, including lattice-based Post-Quantum Cryptography (PQC) schemes (e.g., ML-KEM and ML-DSA). Additionally, Schneider et al. [SPOG19] presented an efficient $B2A_q$ mask conversion algorithm designed for single bits, from which a conversion of arbitrary bits can be derived. Subsequent studies, including those by [BPO⁺20, BDH⁺21, BC22, CGMZ23], have contributed numerous new variants and enhancements, primarily focusing on improving the performance of masked implementations of lattice-based cryptography.

The first higher-order $B2A_{2^k}$ mask conversion algorithm with complexity independent of k was first presented by Hutter and Tunstall [HT16, HT19]. They demonstrated the use of Goubin’s first-order solution to construct a second-order secure $B2A_{2^k}$ mask conversion, requiring only 31 instructions. This concept was subsequently generalized to higher orders in [Cor17, BCZ18] resulting in an algorithm that has an asymptotic runtime complexity of $\mathcal{O}(2^n)$. Recently, Coron et al. [CGTZ23] proposed a solution extending this idea also to prime field operations.

All the aforementioned studies primarily concentrated on secure software implementations, with only a limited number focusing on efficient solutions in hardware. Among the pioneering works, Schneider et al. [SMG15] proposed a hardware implementation of a Boolean masked adder, employing a 3-share Threshold Implementation (TI) and presenting results for both a Ripple-Carry Adder (RCA) and Kogge-Stone (KS) adder. Similar types of masked adders were also presented in [GJM⁺16, BDK⁺21, FBR⁺22, BG22, CGM⁺23, NDKV24] who evaluated other types of adders (e.g., Sklansky adder, Brent-Kung adder, etc.), bit-sliced implementations, or explored different masking techniques (DOM, TI, etc.). It is interesting to note that all of them utilized hardware implementations featuring constructions resembling Boolean masked adders. To the best of our knowledge, no hardware implementation has ever been introduced that performs a **B2A** mask conversion without the need of secure adders. Our work fills this gap by presenting an approach to perform **B2A** mask conversion that bypasses the complexities associated with secure adders, thus paving the way for more efficient hardware implementations of, for instance, lattice-based cryptography. This novel approach not only simplifies the hardware design but also offers improved performance metrics in terms of latency and resource utilization.

Before we explicitly list our contributions, we want to emphasize that in this paper, we aim to improve the performance of **B2A** hardware implementations including execution

time (latency), throughput, and resource utilization. We do not aim to improve the *time complexity* and theoretical upper bound on running-time growth of the underlying B2A algorithms that were proposed in previous works. Instead, we propose new implementations of B2A that effectively arrange and utilize hardware resources to achieve efficient and secure mask-conversion operations. Our goal is to advance state-of-the-art B2A hardware implementations by achieving high performance while maintaining security against side-channel analysis threats.

Our contributions. In this paper, we provide the following contributions:

- We present a first-order secure hardware gadget that performs a $B2A_{2^k}$ mask conversion in a *single cycle*. Previously, achieving this was deemed impractical, as a single-cycle application of Goubin’s algorithm in hardware would lead to exploitable leakage. Our low-latency **10-SecB2A2k** gadget always executes in a single cycle independent of the input word size k . In addition, our new construction has a low area footprint by allowing the use of unmasked additions, avoiding the need for costly Boolean masked adders. For instance, a 32-bit B2A mask conversion requires only 76 LUTs, 94 FFs, and 1 DSP resource on a Kintex-7 FPGA, i.e., an order of magnitude less in LUT and FF requirements compared to related masked Kogge-Stone Adder (KSA) based solutions.
- We introduce a novel second-order secure hardware gadget (**20-SecB2A2k**) capable of conducting $B2A_{2^k}$ mask conversion in just 4 clock cycles. Similarly to our first-order secure solution, our gadget is agnostic to the input word size k and is designed without the need for masked adders. Instead, we exclusively utilize unmasked adders, enabling computation to be delegated to accessible DSPs on FPGAs (or existing adders on ASICs). We present performance results for both *pipelined* and *iterative* versions of the gadget, demonstrating a significant reduction in both latency and resource requirements compared to prior works. Specifically, compared to the latest second-order solution from [BG22], our new gadget reduces latency by a factor of 3 and requires up to 7 times fewer LUTs and 11 times fewer FFs.
- We demonstrate that our novel **SecB2A2k** constructions significantly enhance the performance of mask conversion on integers modulo an arbitrary number $q < 2^k$. Our contribution is twofold: 1) We introduce a new **SecB2Aq** hardware gadget that, unlike previous works [BBE⁺18, FBR⁺22, Cas22], is not based on **SecAdds**. Instead, we utilize our low-latency **SecB2A2k** gadgets to securely convert from Boolean-to-Arithmetic masks while concurrently calculating the single underflow (borrow) bit produced during modulus subtraction, avoiding a full secure addition operation. 2) We compare our solution with the work of Norga et al. [NDKV24]¹, who reported $B2A_q$ performance results for the Kyber (ML-KEM) 12-bit modulus $q = 3329$. Our solution outperforms that work by at least a factor of 3 in terms of speed (latency) and throughput. Furthermore, our gadget reduces resource requirements by more than 87% for first-order security and over 90% for second-order security.
- All our gadgets were proven secure in the robust probing and PINI security model. In order to gain confidence in the security of our gadgets, we also conducted practical DPA tests on a Xilinx Artix-7 FPGA and report the TVLA results. We further provide the Verilog code of our $B2A_{2^k}$ gadgets in the appendix of the paper.

Organization of the paper. The paper is organized as follows. In Section 2, we provide some preliminaries on masking, mask conversion, state-of-the-art security notions, and the Kogge-Stone adder. In Section 3, we discuss first-order secure B2A algorithms and propose

¹In this paper, we refer to the unpublished work [NDKV24] on eprint posted on 26-Jan-2024.

a new $B2A_{2^k}$ and $B2A_q$ gadget which are 1-PINI secure. Section 4 focuses on second-order secure B2A algorithms where we present novel $B2A_{2^k}$ and $B2A_q$ hardware gadgets which are 2-PINI secure. Synthesis results are presented and compared with related work in Section 5. Section 6 presents the results of our practical analysis along with the t-test results. Section 7 concludes the paper.

2 Preliminaries

2.1 Notation

In this paper, we denote the Boolean operations AND, OR, and XOR as \wedge , \vee , and \oplus . We denote a shift of the binary representation of a variable x to the left with $x \ll i$ where i represents the bit positions. Likewise, we denote a right shift by i positions with $x \gg i$. Addition and subtraction operations are represented as $+$ and $-$ signs. We further consider all operations to be performed in the ring \mathbb{Z}_{2^k} where $k \in \mathbb{Z}_{\geq 0}$ represents the bit length of the ring.

We consider a Boolean masking scheme with security order d that splits up a sensitive variable $x \in \mathbb{Z}_{2^k}$ into $d + 1$ Boolean shares (x_1, \dots, x_{d+1}) such that $x = \bigoplus_{i=1}^{d+1} x_i$ to fulfill the *correctness* property. Further note that the shares are selected randomly from a uniform distribution to fulfill the *uniformity* property. A shared function $f(x, y)$ is considered *correct* if the sum of its shared component functions f_i equals to $f(x, y)$, i.e., $f(x, y) = \bigoplus_{i=1}^{d+1} f_i$.

Throughout the work, we use the subscript to refer to the different mask shares. A superscript is used to refer to the individual bits of the mask share. For example, x_3^4 refers to the 4th bit of the third mask share.

2.2 The Glitch-Extended Probing Model

Since the groundbreaking research by Kocher et al. [Koc96], there has been a considerable amount of work on understanding the foundation of Side-Channel Analysis (SCA). Due to sound mathematical background, masking schemes stand out as one of the most widely implemented countermeasures in practice. They have garnered extensive attention from numerous authors in the literature, resulting in various schemes such as Boolean masking [GP99], arithmetic masking [CG00], Inner Product masking [BFG⁺17], or multiplicative masking [GT02]. An essential aspect in the development of these masking schemes is the consideration of adversaries' models, which must account for physical vulnerabilities and diverse execution environments.

The initial proposal for assessing the security of masking schemes was introduced by Ishai et al. [ISW03], known as the “probing model”. In the d th-order probing model, an adversary is capable of observing d intermediate wires of the circuit during cipher execution. A circuit is deemed d^{th} -order secure if any set of d probes remains independent of the secret value, effectively thwarting d^{th} -order side-channel attacks. Put differently, assurance in this model stems from demonstrating that the stochastic values yielded by the probes can be replicated by a simulator lacking access to the circuit’s secret inputs, whereas both the adversary and the authentic circuit possess this information. This security is established through the utilization of randomness generated exclusively by the simulator and the circuit, inaccessible to the adversary. In practical terms, proving probing security involves verifying that the probed values adhere to a random distribution unaffected by the secret value’s selection.

While the probing model serves as an initial security model, practical hardware implementations may still fall short of meeting its security requirements. This is primarily because the model assumes the absence of data-dependent activation, which aligns well

with software platforms, where instructions are executed sequentially. However, in CMOS technologies, a common occurrence known as glitches introduces vulnerabilities. The probing model’s failure to account for the impact of glitches results in insecure designs, as demonstrated in [MPO05, MME10]. Specifically, the d -probing model overlooks physical characteristics such as transitions, coupling effects, or glitches [FGP⁺18]. Therefore, an extended model is necessary to address these undesired effects.

To address the impact of glitches, Reparaz et al. [RBN⁺15] introduced a security model based on the probing model. In this model, a probe on a combinatorial circuit is extended to all signals involved in the computation of the probed wire, extending up to registers or primary inputs. This extension is known as the *glitch-extended* probing model. Later, Faust et al. [FGP⁺18] incorporated this concept into the *robust-probing model*. Consequently, it imposes a significantly stronger security requirement that must be met to ensure the security of designs on hardware platforms. The introduction of this straightforward model enabled relevant scientific communities to develop formal verification tools for evaluating small circuits [KSM20] as the tool is restricted by the complexity limitations of the method for large circuits.

2.3 Masking with $d + 1$ Shares

Threshold Implementation (TI) [NRR06] is one of the first implementation strategies that is secure under the glitch-extended probing model. While TI defines the number of input shares depends on the algebraic degree of the target Boolean function, two separate studies have endeavored to decouple the number of input shares from the algebraic degree, as cited in [RBN⁺15, GMK16]. These studies introduced approaches that utilize $d + 1$ input shares for d^{th} order security, while also upholding glitch resistance on hardware platforms. These studies introduced methods using $d + 1$ input shares for d^{th} order security while ensuring glitch resistance on hardware. Typically, these methodologies require new randomness to ensure non-completeness, contrasting with threshold approaches that might necessitate fresh masks for uniformity. In the $d + 1$ sharing technique, a masked Boolean function is segmented into two distinct sections by a register layer to prevent glitch propagation.

Drawing from the Domain Oriented Masking (DOM) method introduced by the authors of [GMK16], a two-share variant of a two-input AND gate $f(a, b) = ab = y$ can be realized as follows:

$$\begin{array}{rcll}
 f_0(a_0, b_0, c_0, r^1) = a_0 b_0 \oplus c_0 \oplus r^1 & \rightarrow & y'_0 & \\
 f_1(a_0, b_1, r^0) = a_0 b_1 \oplus r^0 & \rightarrow & y'_1 & \\
 \hline
 f_2(a_1, b_0, r^0) = a_1 b_0 \oplus r^0 & \rightarrow & y'_2 & \\
 f_3(a_1, b_1, c_1, r^1) = a_1 b_1 \oplus c_1 \oplus r^1 & \rightarrow & y'_3 &
 \end{array}
 \quad
 \begin{array}{l}
 y'_0 \oplus y'_1 = y_0 \\
 y'_2 \oplus y'_3 = y_1
 \end{array}
 \quad (1)$$

In the given equation in black, denoted as $10\text{-DomAnd}(a_0, a_1, b_0, b_1)$ henceforth, a_0, a_1, b_0, b_1 stand for input shares, r^0 is a single-bit of fresh randomness, and y_0, y_1 are the output shares. The functions f_i are referred to as *component functions*. While it is imperative to store y'_1 and y'_2 in registers, it is optional to do so for y'_0 and y'_3 . The segment responsible for producing output shares through XOR operations on the register outputs is termed the *compression layer*. We can implement the first-order secure mask variant of the AND-XOR function $f(a, b, c) = ab \oplus c$ by incorporating the shares of c , which are c_0 and c_1 , into the first and last component functions, as indicated in blue in Equation 1. This variant is referred to as $10\text{-DomAndXor}(a_0, a_1, b_0, b_1, c_0, c_1)$ in this paper. By introducing an additional random bit r^1 as shown in red in Equation 1, we can refresh all the component functions. Henceforth, we will refer to first-order secure AND and AND-XOR functions with all component functions being refreshed as 10-DomAndRefresh and $10\text{-DomAndXorRefresh}$ throughout the paper. Note that we have omitted mentioning the randomness in the input

list of these gadgets for the sake of simplicity. Likewise, the second-order secure masked AND gate can be realized with the Domain Oriented Masking (DOM) methodology, denoted as $20\text{-DomAnd}(a_0, a_1, a_2, b_0, b_1, b_2)$, requiring only three random bits to mask cross-domain products as follows:

$$\begin{array}{llll}
f_0(a_0, b_0, c_0, r^3, r^4) = a_0b_0 \oplus c_0 \oplus r^3 \oplus r^4 & \rightarrow y'_0 & & \\
f_1(a_0, b_1, r^0, r^3) = a_0b_1 \oplus r^0 \oplus r^3 & \rightarrow y'_1 & y'_0 \oplus y'_1 \oplus y'_2 = y_0 & \\
f_2(a_0, b_2, r^1, r^5) = a_0b_2 \oplus r^1 \oplus r^5 & \rightarrow y'_2 & & \\
\hline
f_3(a_1, b_0, r^0, r^4) = a_1b_0 \oplus r^0 \oplus r^4 & \rightarrow y'_3 & & \\
f_4(a_1, b_1, c_1, r^6, r^7) = a_1b_1 \oplus c_1 \oplus r^6 \oplus r^7 & \rightarrow y'_4 & y'_3 \oplus y'_4 \oplus y'_5 = y_1 & (2) \\
f_5(a_1, b_2, r^2, r^6) = a_1b_2 \oplus r^2 \oplus r^6 & \rightarrow y'_5 & & \\
\hline
f_6(a_2, b_0, r^1, r^8) = a_2b_0 \oplus r^1 \oplus r^8 & \rightarrow y'_6 & & \\
f_7(a_2, b_1, r^2, r^7) = a_2b_1 \oplus r^2 \oplus r^7 & \rightarrow y'_7 & y'_6 \oplus y'_7 \oplus y'_8 = y_2 & \\
f_8(a_2, b_2, c_2, r^5, r^8) = a_2b_2 \oplus c_2 \oplus r^5 \oplus r^6 & \rightarrow y'_8 & &
\end{array}$$

By placing the shares of c in Equation 2, as marked in blue, we can realize a 2nd-order secure version of the AND-XOR function, which we name $20\text{-DomAndXor}(a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2)$ in this paper. Similarly, we can refresh all component functions in a circular approach as discussed in [RBN⁺15] with nine random bits in total as denoted in red in Equation 2, creating 20-DomAndRefresh and $20\text{-DomAndXorRefresh}$. For further information on the second-order masked AND gate, we refer to the original publication [GMK16] to keep this section concise.

2.4 Security Notions

While the glitch-extended probing model allows for the evaluation of a given circuit, it falls short in guaranteeing the secure composability of gadgets. In practical circuits, masking an entire cipher or circuit poses significant challenges, necessitating a modular divide-and-conquer strategy. Consequently, rather than tackling the entire circuit at once, we break it down into smaller building blocks and construct a protected circuit by assembling these smaller protected units. This approach requires meeting a set of criteria to ensure the security of the composed circuit under the glitch-extended probing model. These small building blocks can be simple logic gates like AND and XOR, or larger circuits, which we refer to as *gadgets* in this paper. The theoretical framework for achieving composability and security through these gadgets is referred to as *composability notions*, which establish the conditions for ensuring the secure interconnection of the gadgets. Over time, several composability notions have been introduced, incorporating conservative assumptions to ensure that glitch-extended probing security remains assured regardless of how these gadgets are interconnected.

One of the earliest notions, introduced in [BBD⁺15], is known as Non-Interference (NI). It aims to limit probe propagation within the gadget to ensure security in composition. Put simply, this notion seeks to guarantee that any probe within the gadgets only extends into a restricted set of input shares. In this notion, any set of $t \leq d$ probes observes a maximum of t input shares. However, it has been demonstrated that ensuring secure composition of gadgets requires more than just the NI notion. Consequently, a new notion called Strong Non-Interference (SNI) [BBD⁺16] has been introduced, which imposes additional restrictions, making the implementation costs higher. For example, in Boolean masking, applying linear functions straightforwardly to each share individually is not feasible under this notion. Extra care must be taken to ensure compliance with SNI, usually achieved by introducing additional fresh masks, which adds more overhead to the design.

Algorithm 1: Carry-vector calculation using a Kogge-Stone adder

Input: Integers a, b where $a, b \in \mathbb{Z}_{2^k}$
Output: Carry vector $c = (a + b) \oplus a \oplus b$ with $c \in \mathbb{Z}_{2^{k+1}}$

```

1  $g^0 \leftarrow a^0 \wedge b^0$ 
2  $p^0 \leftarrow a^0 \oplus b^0$ 
3 for  $i := 1$  to  $k - 1$  do
4    $g^i \leftarrow (b^{i-1} \wedge a^{i-1} \wedge a^i) \oplus (b^i \wedge a^i) \oplus (b^{i-1} \wedge b^i \wedge a^{i-1})$ 
5    $p^i \leftarrow (b^{i-1} \oplus a^{i-1}) \wedge (b^i \oplus a^i)$ 
6 end
7 for  $j := 2$  to  $w = \lceil \log_2(k - 1) \rceil - 1$  do
8    $\beta \leftarrow 2^{j-1}$ 
9    $g \leftarrow g \oplus (p \wedge (g \ll \beta))$ 
10   $p \leftarrow p \wedge (p \ll \beta)$ 
11 end
12  $\beta \leftarrow 2^w$ 
13  $g \leftarrow g \oplus (p \wedge (g \ll \beta))$ 
14  $c \leftarrow g \ll 1$ 
15 return  $c$ 

```

This issue was addressed in [CS20], where the notion of Probe-Isolating Non-Interferenc (PINI) was introduced. This notion carries some similarity with DOM [GMK16] as the author also introduced share domains. In general, under this notion, any probe placed on a gadget is restricted to propagating within a specific single share domain, and it is not allowed to observe different domains with the same probe. If the probe is positioned at the output shares, it must be limited to propagating within the same share domain as the output share on which the probe was placed. Hence, this notion has the advantage of the trivial realization of linear functions in Boolean masking reducing the overhead of implementations compared to SNI. As we mainly focus on PINI notion, we restate the definition of PINI below borrowed from [Kni23].

Definition 1 (*d-Probe Isolating Non-Interference (PINI)*). Let \mathcal{P}_I be a set of t_1 internal probes and \mathcal{P}_O be a set of t_2 output wire probes, where \mathcal{I}_O is the index set associated with the probes on the output wires. A masked circuit \mathcal{G} is d-PINI iff for every $\mathcal{P} = \mathcal{P}_I \cup \mathcal{P}_O$ with $t_1 + t_2 \leq d$, there exists a set \mathcal{I}_I of circuit indices with $|\mathcal{I}_I| \leq t_1$ such that \mathcal{P} can be perfectly simulated by input shares with indices drawn only from $\mathcal{I}_I \cup \mathcal{I}_O$.

Drawing from the insights presented in [CGLS21] and leveraging the *refresh-then-multiply* technique outlined therein, refreshing one of the input variables before passing it through the DOM multiplication forms a PINI gadget, referred to as Hardware Private Circuits 1 (HPC1). In this paper, we also employ *refresh-then-multiply* technique to ensure PINI security of our designs.

2.5 Kogge-Stone Adder

The Kogge-Stone Adder (KSA) is a parallel prefix adder used in digital circuit design for performing fast addition [KS73]. One of the key features of the Kogge-Stone adder is its efficient carry propagation mechanism. Unlike Ripple Carry Adders (RCA), where the carry must propagate sequentially through each bit position, the Kogge-Stone adder uses a parallel carry network that enables simultaneous carry computation across multiple bit positions. Note that the Kogge-Stone adder exhibits excellent scalability. As the number of bits in the operands increases, the depth of the binary tree grows logarithmically rather than linearly, resulting in a more efficient use of resources and improved performance.

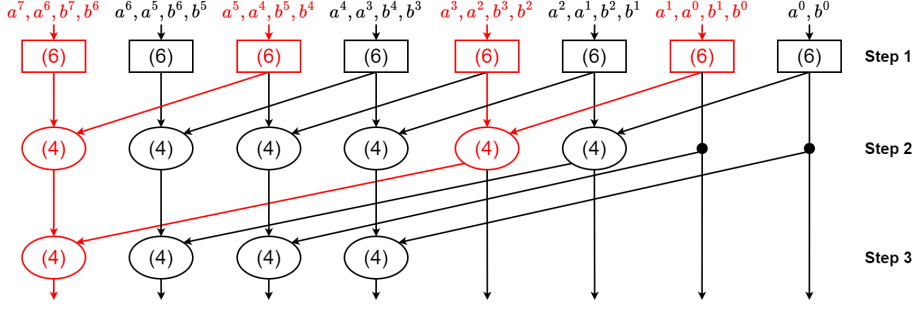


Figure 1: The structure of the carry generation for two 8-bit inputs using a Kogge-Stone Adder (KSA). Operations specifically related to the last carry bit are highlighted in red.

In this algorithm, the carry bits for two k -bit inputs a and b are computed by a pre-processing step and a prefix-computation step. In the pre-processing, the initial p^i and g^i values are calculated as

$$g^i = a^i \wedge b^i, \quad p^i = a^i \oplus b^i, \quad (3)$$

where a^i and b^i are the i^{th} bit of the input values a and b . In the prefix-computation step, the carry bits are computed in $\log_2(k)$ stages. In each stage, the group carry propagate $p^{i:j}$ and group generate signals $g^{i:j}$ are generated for $i > j$ as

$$g^{i:j} = g^i \oplus (g^j \wedge p^i), \quad p^{i:j} = p^i \wedge p^j. \quad (4)$$

After $\log_2(k)$ stages, all carry bits can be derived as

$$c^1 = g^0, \quad c^{i \in \{2 \dots k\}} = g^{i-1:0}. \quad (5)$$

In this paper, we combine the pre-processing step and the first stage of the prefix-computation. Hence, the output of the first stage can be calculated directly as follows:

$$g^i = \begin{cases} a^i \wedge b^i & \text{if } i = 0 \\ (b^{i-1} \wedge a^{i-1} \wedge a^i) \oplus (b^i \wedge a^i) \oplus (b^{i-1} \wedge b^i \wedge a^{i-1}) & \text{otherwise} \end{cases} \quad (6)$$

$$p^i = \begin{cases} a^i \oplus b^i & \text{if } i = 0 \\ (b^{i-1} \oplus a^{i-1}) \wedge (b^i \oplus a^i) & \text{otherwise} \end{cases}$$

The remaining calculations proceed as previously described, employing Equation 4 and Equation 5, as depicted in Figure 1 for two 8-bit inputs. Algorithm 1 outlines the algorithm for computing all carry bits, i.e., the carry vector.

If only the *overflow* bit, i.e., the most significant carry bit c^{k+1} of the carry vector $c \in \mathbb{Z}_{2^k}$, is needed, we can trivially simplify the computation, as not all intermediate values contribute to the overflow bit c^{k+1} . For instance, if we are only interested in determining whether $X > Y$, we can simply calculate $X - Y$ and check the overflow bit.

As an example, let's consider $k = 2^3$. As demonstrated in Figure 1, highlighted in red, only a few nodes need to be computed. In fact, only half of the nodes need to be computed using Equation 6 in Step 1, two nodes need to be computed in Step 2, and 1 node in Step 3. That means, in general, for $k = 2^n$, the number of nodes that need to be computed in Step i is defined as $\frac{k}{2^i}$. Algorithm 2 shows the algorithm to calculate only the *overflow* bit of an integer addition.

Algorithm 2: Carry-bit calculation using a Kogge-Stone adder

Input: Integers a, b where $a, b \in \mathbb{Z}_{2^k}$
Output: Carry bit $c = (((a + b) \oplus a \oplus b) \ggg k)$ with $c \in \mathbb{F}_2$

- 1 **for** $i := 1$ **to** $k - 1$ **by** 2 **do**
- 2 $g^i \leftarrow (b^{i-1} \wedge a^{i-1} \wedge a^i) \oplus (b^i \wedge a^i) \oplus (b^{i-1} \wedge b^i \wedge a^{i-1})$
- 3 $p^i \leftarrow (b^{i-1} \oplus a^{i-1}) \wedge (b^i \oplus a^i)$
- 4 **end**
- 5 **for** $j := 2$ **to** $\log_2(k) - 1$ **do**
- 6 $\alpha \leftarrow k$; $\beta \leftarrow 2^{j-1}$
- 7 **for** $i := 1$ **to** $\frac{k}{2^j}$ **do**
- 8 $m \leftarrow \alpha - 1$
- 9 $l \leftarrow \alpha - \beta - 1$
- 10 $t \leftarrow p^m$
- 11 $p^m \leftarrow t \wedge p^l$
- 12 $g^m \leftarrow g^m \oplus (t \wedge g^l)$
- 13 $\alpha \leftarrow \alpha - 2\beta$
- 14 **end**
- 15 **end**
- 16 $c \leftarrow g^{k-1} \oplus (p^{k-1} \wedge g^{k/2-1})$
- 17 **return** c

3 First-order Secure B2A Mask Conversion

We will now discuss and present first-order secure Boolean-to-Arithmetic (B2A) mask conversion algorithms. First, we discuss mask conversions applied on integers over \mathbb{Z}_{2^k} (B2A_{2^k}). We propose a novel gadget that can perform the conversion efficiently in hardware. After that, we propose a new hardware gadget that performs a mask conversion on integers modulo an arbitrary number $q < 2^k$ (B2A_q).

3.1 Goubin's Solution

In 2001, Goubin proposed an efficient method of converting a Boolean mask to an arithmetic mask that is first-order probing secure [Gou01]. The method is independent of the input word size k and requires a constant number of instructions.

The essential observation of Goubin was that the function

$$\Phi_{\mathbb{Z}}(a, b) : \mathbb{Z}^2 \longrightarrow \mathbb{Z} : a, b \longmapsto (a \oplus b) + b \quad (7)$$

is affine over \mathbb{F}_2 . That means that $(\Phi(a, b) \oplus \Phi(a, 0))$ is *linear* for any $b \in \mathbb{Z}$. We can further note that the same function is applicable in the field $(\mathbb{Z}_{2^k}, \oplus, +)$, for any $k \in \mathbb{Z}_{\geq 0}$. We can therefore define the function as follows:

$$\begin{aligned} \Phi(a, b) : (\mathbb{Z}_{2^k}, \oplus, +)^2 &\longrightarrow (\mathbb{Z}_{2^k}, \oplus, +) \\ a, b &\longmapsto (a \oplus b) + b \end{aligned} \quad (8)$$

for some $k \in \mathbb{Z}_{\geq 0}$.

We can now use this affine function to perform a secure mask conversion. Let's consider $x' = x \oplus r$, where $x, r \in (\mathbb{Z}_{2^k}, \oplus, +)$ and x' represents the Boolean masked secret x and r represents a random value taken from \mathbb{Z}_{2^k} . We wish to be able to remove the Boolean mask and replace it with an arithmetic mask instead by applying the function to x' , i.e., $\Phi(x', r) = (x' \oplus r) + r = x''$, where $x'' = x + r$ represents the arithmetically masked secret x . However, this operation would reveal information on x through some side channels. To

Algorithm 3: Goubin's 1st-Order Secure B2A_{2^k} Mask Conversion**Input:** $x' = x \oplus r$, the mask r , a random integer γ , where $x, r, \gamma \in (\mathbb{Z}_{2^k}, \oplus, +)$ **Output:** $x'' = x + r$

1 $t \leftarrow x' \oplus \gamma$ 2 $t \leftarrow t + \gamma$ 3 $t \leftarrow t \oplus x'$ 4 $\gamma \leftarrow \gamma \oplus r$	5 $z \leftarrow x' \oplus \gamma$ 6 $z \leftarrow z + \gamma$ 7 $z \leftarrow z \oplus t$ return z
---	---

avoid this, we can mask the computation by adding an additional random value $\gamma \in \mathbb{Z}_{2^k}$ and compute

$$\Phi(x', \gamma \oplus r) = (x' \oplus (\gamma \oplus r)) + (\gamma \oplus r), \quad (9)$$

which can then be followed by an unmasking step using

$$\Phi(x', \gamma) = (x' \oplus \gamma) + \gamma. \quad (10)$$

Therefore, we can perform a secure Boolean-to-arithmetic mask conversion as follows:

$$\begin{aligned} x'' &= x' \oplus \Phi(x', \gamma) \oplus \Phi(x', \gamma \oplus r) \\ &= x' \oplus [(x' \oplus \gamma) + \gamma] \oplus [(x' \oplus (\gamma \oplus r)) + (\gamma \oplus r)]. \end{aligned} \quad (11)$$

One can implement this conversion using 7 instructions: 2 additions and 5 XOR operations as shown in Algorithm 3. In the following, we aim to construct an efficient hardware implementation of Goubin's B2A algorithm.

3.2 First-order Secure Realization of B2A_{2^k} in Hardware

Let $x' = x \oplus r$ and r be the Boolean sharing of the sensitive variable x . In a first step, we need to refresh the input shares with two random variables γ and β to achieve composable security. We can define an initial sharing $a = x$ with three shares (a_1, a_2, a_3) as follows:

$$a_1 = x' \oplus \gamma, \quad a_2 = r \oplus \gamma \oplus \beta, \quad a_3 = \beta.$$

The uniformly distributed shares are then stored in separate registers. Then, we can define an intermediate sharing b with shares (b_1, b_2, b_3) , i.e.,

$$\begin{aligned} b_1 &= \Phi(a_1, a_3) \oplus a_1 = ((a_1 \oplus a_3) + a_3) \oplus a_1, \\ b_2 &= \Phi(a_1, a_2) = ((a_1 \oplus a_2) + a_2), \\ b_3 &= a_2 \oplus a_3. \end{aligned}$$

Finally, we can compute the arithmetically masked output z using 2 shares (z_1, z_2) where

$$\begin{aligned} z_1 &= b_1 \oplus b_2, \\ z_2 &= b_3, \end{aligned}$$

and $x = z_1 - z_2$. Gadget 1 shows a hardware description of this mask-conversion solution. It is compatible with any input word size k and requires a fixed number of 2 clock cycles to perform a mask conversion operation. Two fresh random variables γ and β are needed for the beginning. Note that the solution can process inputs in each cycle and therefore supports pipelining. We now analyze our gadget in terms of side-channel resistance security.

Gadget 1: 1-PINI Secure B2A_{2^k} Mask Conversion based on Goubin's Solution

Input: Shares $x' = x \oplus r$, the mask r , two random integers γ and β , where
 $x, r, \gamma, \beta \in (\mathbb{Z}_{2^k}, \oplus, +)$

Output: $z = (z_1, z_2)$ where $x = z_1 - z_2$

Cycle 1:

2 $a_1 \leftarrow x' \oplus \gamma$

3 $a_2 \leftarrow r \oplus \gamma \oplus \beta$

4 $a_3 \leftarrow \beta$

Cycle 2:

6 $b_1 \leftarrow ((a_1 \oplus a_3) + a_3) \oplus a_1$

7 $b_2 \leftarrow ((a_1 \oplus a_2) + a_2)$

8 $b_3 \leftarrow a_2 \oplus a_3$

Outputs:

10 $z_1 \leftarrow b_1 \oplus b_2$

11 $z_2 \leftarrow b_3$

return $z = (z_1, z_2)$

Theorem 1. *Gadget 1 is first-order glitch-robust PINI.*

Proof. As discussed in Section 2.4 and Definition 1, proving the security of the gadgets under PINI notions requires demonstrating that any probe on the intermediate values (internal probes) extends only into a single arbitrary share domain, and any probe on the output wires propagates at most to the same share domain as the probed output. Probing any initial sharing a_i would reveal information about only one share domain. Probing b_1 would yield information about a_1 and a_3 , where the former is blinded by γ and the latter is the fresh randomness β . Probing b_2 would provide information about a_1 and a_2 , which are blinded by γ and β , respectively. Probing b_3 would result in information about a_2 and a_3 , which are also blinded by γ and β , respectively. Hence, probing any intermediate values b_i satisfies the 1-PINI requirements. The probe placed on the output share z_2 results in information about b_3 which is completely blinded by γ . Probing the output share z_1 would result in revealing information about b_1 and b_2 . Each bit of b_2 can be calculated as:

$$b_2^i = \begin{cases} a_1^0 & \text{if } i = 0 \\ a_1^i \oplus f_i(\{a_1^j, 0 \leq j \leq i-1\}, \{a_2^j, 0 \leq j \leq i-1\}) & \text{otherwise} \end{cases}$$

where the function $f_i(\cdot)$ has the algebraic degree of $i+1$. Note that all b_i are stored in a register, so the extended probe to b_2^i can reveal information at most about a_1^i . In other words, it can be interpreted that each function $f_i(\cdot)$ is masked by a_1^i in the computation of b_2^i as a_1^i is not in the $f_i(\cdot)$ input lists. b_1 is a function of a_1 and a_3 , and since the extended probe on b_2^i reveals information about a_1^i , it is possible to gain information about some bits of a_3 . As defined in the beginning of this section, a_3 is simply the random value β , and a_1 is blinded by γ . Hence, as previously mentioned, a_1 and a_3 are independent of each other, with a_1 being blinded by γ . Thus, probing the output share z_1 fulfills the 1-PINI requirements. Since all internal probes and probes placed on output shares meet 1-PINI requirements, Gadget 1 is 1-PINI secure. \square

We also verified the security of Gadget 1 in the PINI security model using the formal verification tool SILVER [KSM20]. SILVER is designed to check hardware designs against formal proofs, eliminating the need to manually write proofs for each design. It evaluates the gate-level netlist of a hardware design and reports results based on security notions defined in various studies, such as [MBR19]. Since SILVER does not perform any simplifications, its analysis results are reliable and free of false positives or negatives. To this end, we synthesized the HDL code of our gadget and provided SILVER with the resulting netlist for evaluation. As a result, SILVER confirmed that the gadget is first-order glitch-robust PINI.

3.3 A Low-Latency First-Order Secure B2A_{2^k} Gadget Proposal

We can reduce the complexity of our composable secure Boolean-to-arithmetic mask conversion from the previous subsection as follows.

Let's recall the computation of the arithmetic output sharing $z = (z_1, z_2)$, i.e.,

$$\begin{aligned} z_1 = b_1 \oplus b_2 &= \Phi(a_1, a_3) \oplus a_1 \oplus \Phi(a_1, a_2) \\ &= [\Phi(x' \oplus \gamma, \beta) \oplus x' \oplus \gamma] \oplus [\Phi(x' \oplus \gamma, r \oplus \gamma \oplus \beta)], \\ z_2 = b_3 &= a_2 \oplus a_3 = r \oplus \gamma. \end{aligned}$$

It is evident that z_2 remains independent of the secret x , allowing for secure computation within a single cycle. However, for z_1 , directly computing $\Phi(a_1, a_2)$ would compromise x 's secrecy, necessitating a latency increase by 1 to flop one input before the computation. Additionally, computing $b_1 \oplus b_2$ simultaneously would result in a first-order leak because all three shares of a would get combined in the same share domain. Thus, an additional register layer is required before determining the output share z_1 . Consequently, this gadget requires a minimum latency of two clock cycles for computation.

In order to perform the mask conversion in a single cycle, we propose to eliminate one of the Φ computations. Let's define new intermediate values (a_1, a_2, a_3, a_4) as follows:

$$\begin{aligned} a_1 &= x', & a_3 &= r + \gamma, \\ a_2 &= x' \oplus \gamma, & a_4 &= r \oplus \gamma. \end{aligned}$$

Then, we can directly compute the output sharing z , i.e.,

$$\begin{aligned} z_1 &= \Phi(a_4, a_1) \oplus a_3 \oplus a_4, \\ z_2 &= a_2, \end{aligned}$$

with $x = z_1 - z_2$ and $z_1 = x + (x' \oplus \gamma)$ and $z_2 = x' \oplus \gamma$. Note that our new gadget does only contain one secret-value dependent Φ computation. Further note that the terms a_3 and a_4 only depend on the random values r and γ and can be calculated in the same share domain.

It is important to note that the computation of z_1 will leak information of x when the output of the term $\Phi(a_4, a_1)$ gets XORed with the remaining term $a_3 \oplus a_4$. This is because the term $a_3 \oplus a_4$ represents the carries $c_{r+\gamma}$ produced by the addition of $r + \gamma$, i.e.,

$$c_{r+\gamma} = a_3 \oplus a_4 = (r + \gamma) \oplus r \oplus \gamma$$

and the carries, which depend on bits of r and γ , will unmask x during the computation of $\Phi(a_4, a_1) = \Phi(r \oplus \gamma, x') = r \oplus \gamma \oplus x' + x' = x \oplus \gamma + x \oplus r$. However, we can observe that the term a_4 is used twice in the computation of z_1 . Thus, we can eliminate this term during calculation of $\Phi(a_4, a_1)$, which pleasantly also avoids unmasking the sensitive variable x .

Gadget 2 shows our proposed mask-conversion solution. First, we compute all shares a_i for $i = 1..4$ (Line 1) and store the results in registers. Second, we calculate z_1 securely (Lines 2-13) using a modified Ripple-Carry Adder (RCA). The construction does eliminate the terms a_4 from the carry-chain computation and therefore does not cause any leakage of x . Note that z_1 can be computed in the same share domain and therefore in a single clock cycle. The Verilog code based on this algorithm is provided in [Appendix B](#).

Theorem 2. *Gadget 2 is first-order glitch-extended PINI (1-PINI).*

Proof. Similar to Theorem 1, we first demonstrate that any internal probe propagates to at most one input share. Examining Line 1 in Gadget 2, we observe that placing any

Gadget 2: 10-SecB2A2k: 1-PINI Secure B2A_{2^k} Mask Conversion**Input:** Shares $x' = x \oplus r$, the mask r , a random integer γ , where $x, r, \gamma \in (\mathbb{Z}_{2^k}, \oplus, +)$ **Output:** $z = (z_1, z_2)$ where $x = z_1 - z_2$ with $z_1 = x + (x \oplus r \oplus \gamma)$ and $z_2 = x \oplus r \oplus \gamma$

```

1  $a_1 \leftarrow x'$  ;  $a_2 \leftarrow x' \oplus \gamma$  ;  $a_3 \leftarrow r + \gamma$  ;  $a_4 \leftarrow r \oplus \gamma$ 
2  $z_1^0 \leftarrow a_3^0$  ;  $z_1^1 \leftarrow a_3^1 \oplus (a_1^0 \wedge \neg a_4^0)$ 
3 for  $i := 2$  to  $k - 1$  do
4    $z_1^i \leftarrow a_3^i \oplus (a_1^{i-1} \wedge \neg a_4^{i-1})$ 
5   for  $d := 3$  to  $i + 1$  do
6      $j \leftarrow i + 1 - d$ 
7      $t \leftarrow a_1^j \wedge \neg a_4^j$ 
8     for  $m := d - 2$  to  $1$  do
9        $t \leftarrow t \wedge a_4^{j+m}$ 
10    end
11     $z_1^i \leftarrow z_1^i \oplus t$ 
12  end
13 end
14  $z_2 \leftarrow a_2$ 
15 return  $z = (z_1, z_2)$ 

```

probe on the calculation of intermediate values a_i extends to at most one input share, thus satisfying the 1-PINI requirements. Next, we show that any probe on output shares would result in propagation of the probe with same share domain. Let us assume that x' and z_1 are in the share domain A and r and z_2 are in the share domain B. Looking at Lines 2 to 13 in Gadget 2, we can conclude that probing z_1^i would result in gaining information about a_3^i , $\{a_1^j, 0 \leq j < i\}$, and $\{a_4^j, 0 \leq j < i\}$. Further note that a_3^i is blinded by γ^i , $\{a_4^j, 0 \leq j < i\}$ are blinded by $\{\gamma^j, 0 \leq j < i\}$, and $\{a_1^j, 0 \leq j < i\}$ are basically revealing information about the input share x' which is in the same domain as z_1 , fulfilling the requirements of 1-PINI. Showing the security of probing z_2 is trivial as it revealed information about a_2 which is blinded by γ . Hence, Gadget 2 is first-order glitch-robust PINI (1-PINI). \square

In addition to the theoretical proof, we verified the first-order glitch-robust PINI security of Gadget 2 using the formal verification tool SILVER [KSM20].

3.4 A Novel First-Order Secure B2A_qGadget Proposal

Many cryptographic algorithms require to perform a B2A mask conversion in the field of integers modulo an arbitrary number $q < 2^k$. That means that the Boolean masked secret x must be reduced modulo q before it gets arithmetically shared. One way to perform the modulo reduction is to select either $x - q$ in case $x - q \geq 0$ or x in the case $x - q < 0$, i.e.,

$$x \bmod q = \begin{cases} x - q & \text{if } x - q \geq 0 \\ x & \text{if } x - q < 0. \end{cases}$$

This method was first proposed by Barthe et al. [BBE⁺18]. They proposed to add the two's complement of q to x with the help of a Boolean masked adder (**SecAdd**) gadget. The generated carry bit is then used to select the output of a masked multiplexer, which can be constructed using two **SecAnd** gadgets.

Instead of using a Boolean masked adder, we propose to use our 10-SecB2A2k gadget in combination with a novel gadget, further called 10-KSABorrowBitGen. This gadget securely calculates the borrow bit of a subtraction operation using a Boolean masked carry chain. However, in contrast to a classical Boolean masked adder, only the masked borrow bit is output instead of the full Boolean masked sum of the inputs. This allows for an optimized implementation as only the underflow bit (the borrow) needs to be computed.

Gadget 3 outlines the general structure of calculating the borrow bit b in a masked format. We utilize the *refresh-then-multiply* technique to ensure the gadget’s 1-PINI security as discussed in [CGLS21]. The gadget takes the Boolean shares of $x = (x_0, x_1)$ and the constant value q . Looking at Lines 2 and 3 in Algorithm 2, we notice that the calculations of p^i and g^i are quadratic, as the second operand of the adder is a constant. Moreover, only the odd bits of p^i and g^i are necessary, each being a function of two adjacent input bits, $i - 1$ and i . Therefore, we refresh only the even bits of (x_0, x_1) using the **Refresh** gadget, XORing a single-bit fresh mask to each bit, as shown in Lines 1-3 in Gadget 3. More precisely, the **Refresh** gadget takes two shares (x_0^i, x_1^i) and refresh them with a single bit fresh masks r^0 generating output shares $(x_0^i \oplus r^0, x_1^i \oplus r^0)$. Then, we compute $x - q = x + (2^k - q) = x + u$ to generate the desired borrow bit based on Algorithm 2. As mentioned before, the functions to initialize p and g are quadratic and we can make use of the DOM approach to realize first-order secure masked variation in the first step. We employ **10-DomAnd**, described in Subsection 2.3, to generate (p_0, p_1) , requiring a single-bit fresh mask for each output bit. We utilize **10-DomAndRefresh** to generate (g_0, g_1) , which requires 2-bit randomness for each output bit. Note that u^i and u^{i-1} are constant bits, thus there is no need to share them. They are provided to the **10-DomAndRefresh** gadget solely for calculating the correct output. Both **10-DomAnd** and **10-DomAndRefresh** are performed in a single cycle and we do not place any register after the compression layer. Hence, it is necessary to refresh all component functions in the calculation of (g_0^i, g_1^i) using **10-DomAndRefresh** to meet 1-PINI requirements when the output shares are given to the next masked function. Likewise, we follow the same approach in the calculation of next steps outlined in Lines 12 to 24 since all functions in the next steps are quadratic. More precisely, we use **10-DomAnd** to generate (p_0, p_1) and **10-DomAndXorRefresh** to calculate (g_0, g_1) in the next steps. We should highlight that the calculation of (g_0, g_1) is in the form of **AND-XOR** function and hence we have to use **10-DomAndXorRefresh** (see Line 12 in Algorithm 2).

Using these gadgets, the mask conversion can be performed as follows:

1. We compute a B2A mask conversion of the Boolean input $x' = x \oplus r$ using **10-SecB2A2k** and obtain the two arithmetic shares $t_1 = x + (x' \oplus \gamma)$ and $t_2 = (x' \oplus \gamma)$, where $x = t_1 - t_2$.
2. We calculate $u = (x' \oplus \gamma) + q$ by adding the modulus q to the second output share t_2 .
3. We compute the masked borrow bit $b = b_1 \oplus b_2$ that is generated during the subtraction of $x - p$ using a masked borrow-bit generator gadget (**10-KSABorrowBitGen**) that is based on a simplified carry-lookahead adder.
4. The first output share z_1 is set to t_1 . The second output share is selected as follows: depending on the masked borrow bit b , we either take t_2 or u as the second output share z_2 , i.e.,

$$z_2 = \begin{cases} t_2 = x' \oplus \gamma & \text{if } b = 0 \\ t_3 = (x' \oplus \gamma) + q & \text{if } b = 1. \end{cases}$$

The final output sharing z is then composed of the two arithmetic shares z_1 and z_2 where $x = z_1 - z_2$ and $x \in [0, q - 1]$.

Gadget 4 shows a hardware description of the proposed solution. It makes use of the composable gadgets Gadget 2 (**10-SecB2A2k**) and Gadget 3 (**10-KSABorrowBitGen**). We denote by \tilde{b} the extension of the bit b to the entire k -bit word. These bits get used in the masked multiplexer (Lines 4-5). In terms of performance, since all gadgets that we used including **10-DomAnd**, **10-DomAndRefresh**, and **10-DomAndXorRefresh** are performed in a single cycle the latency of Gadget 3 for k -bit input variables is $\log_2(k) + 1$. Note that, one extra cycle is required for refreshing the input and $\log_2(k)$ is needed to calculate the borrow

Gadget 3: 10-KSABorrowBitGen: 1-PINI Secure Borrow-Bit Generator

Input: Shares $x = (x_0, x_1)$, where $x_0, x_1 \in \mathbb{Z}_{2^k}$, modulus $q < 2^k$.
Output: Borrow bit $b = (b_0, b_1)$ generated during $x - q$ where $b_0, b_1 \in \mathbb{F}_2$

```

1 for  $i := 0$  to  $k - 2$  by 2 do
2    $(a_0^i, a_1^i) \leftarrow \text{Refresh}(x_0^i, x_1^i)$ 
3 end
4 for  $i := 1$  to  $k - 1$  by 2 do
5    $(a_0^i, a_1^i) \leftarrow (x_0^i, x_1^i)$ 
6 end
7  $u \leftarrow 2^k - q$ 
8 for  $i := 1$  to  $k - 1$  by 2 do
9    $(p_0^i, p_1^i) \leftarrow \text{10-DomAnd}(a_0^i \oplus u^i, a_1^i, a_0^{i-1} \oplus u^{i-1}, a_1^{i-1})$ 
10   $(g_0^i, g_1^i) \leftarrow \text{10-DomAndRefresh}(a_0^i, a_1^i, a_0^{i-1}, a_1^{i-1}, u^i, u^{i-1})$ 
11 end
12 for  $j := 2$  to  $\log_2(k) - 1$  do
13    $\alpha \leftarrow k$ 
14    $\beta \leftarrow 2^{j-1}$ 
15   for  $i := 1$  to  $\frac{k}{2^j}$  do
16      $m \leftarrow \alpha - 1$ 
17      $l \leftarrow \alpha - \beta - 1$ 
18      $(t_0, t_1) \leftarrow (p_0^m, p_1^m)$ 
19      $(p_0^m, p_1^m) \leftarrow \text{10-DomAnd}(t_0, t_1, p_0^l, p_1^l)$ 
20      $(g_0^m, g_1^m) \leftarrow \text{10-DomAndXorRefresh}(t_0, t_1, g_0^l, g_1^l, g_0^m, g_1^m)$ 
21      $\alpha \leftarrow \alpha - 2\beta$ 
22   end
23 end
24  $(b_0, b_1) \leftarrow \text{10-DomAndXorRefresh}(p_0^{k-1}, p_1^{k-1}, g_0^{\frac{k}{2}-1}, g_1^{\frac{k}{2}-1}, g_0^{k-1}, g_1^{k-1})$ 
25 return  $(b_0, b_1)$ 

```

bit. The latency of Gadget 2 is 1, therefore our Gadget 4 has a complexity of $\mathcal{O}(\log k)$ and needs $\log_2(k) + 2$ clock cycles to perform the mask conversion, i.e., 7 cycles for a 32-bit B2A. The gadget is pipelineable and needs 48 bits of fresh randomness in each cycle.

Theorem 3. *Gadget 4 is first-order glitch-robust PINI (1-PINI).*

Proof. We first discuss the 1-PINI security of the 10-KSABorrowBitGen gadget. We use the *refresh-then-multiply* technique as mentioned in [CGLS21]. In short, this technique involves using the DOM methodology with one input being refreshed, which is shown to be PINI secure. We follow this technique in Gadget 3 as well. In Line 2, we refresh one of the inputs to the DOM AND gates in Lines 9 and 10. We use `10-DomAndRefresh` to generate (g_0^i, g_1^i) to ensure that one of the AND inputs is refreshed in Line 20, where `10-DomAndXorRefresh` is used to keep (g_0^i, g_1^i) always refreshed. Note that using `10-DomAndRefresh` is not necessary to calculate (p_0^i, p_1^i) in Lines 9 and 19, as the refreshing in Line 2 is sufficient to satisfy 1-PINI requirements in the entire design. Namely, since one of the inputs to the `10-DomAnd` gadget is already refreshed in Line 2, any probe in calculating the (p_0^i, p_1^i) tree in the KSA algorithm is always expanded to the same share domain, thereby ensuring 1-PINI security. We confirmed the 1-PINI security of 10-KSABorrowBitGen using SILVER as well.

Since both 10-SecB2A2k and 10-KSABorrowBitGen are 1-PINI secure, it is enough to show that probing Line 2 and Lines 4-7 meets 1-PINI requirements in Gadget 4. Since q is a constant, probing u would reveal the same information as t_2 . As t_2 is the output of a 1-PINI secure gadget, proving the 1-PINI security is trivial. Probing Line 4 (w_1) would

Gadget 4: 10-SecB2Aq: 1-PINI Secure B2A_q Mask Conversion

Input: Shares $x' = x \oplus r$, the mask r , where $x, r \in \mathbb{Z}_{2^k}$, modulus $q < 2^k$.

Output: $z = (z_1, z_2)$ where $x = z_1 - z_2$ with $x \in [0, q - 1]$

```

1  $(t_1, t_2) \leftarrow \text{10-SecB2A2k}(x', r)$ 
2  $u \leftarrow t_2 + q$ 
3  $(b_1, b_2) \leftarrow \text{10-KSABorrowBitGen}(x', r, q)$ 
4  $w_1 \leftarrow (\tilde{b}_1 \wedge (t_2 \oplus u)) \oplus t_2$ 
5  $w_2 \leftarrow (\tilde{b}_2 \wedge (t_2 \oplus u))$ 
6  $z_1 \leftarrow t_1$ 
7  $z_2 \leftarrow w_1 \oplus w_2$ 
8 return  $z = (z_1, z_2)$ 

```

reveal information about b_1 and t_2 (u has the same information as t_2). Similarly, probing Line 5 (w_2) would reveal information about b_2 and t_2 . Since (t_1, t_2) and (b_1, b_2) are output shares of two different 1-PINI gadgets, we can conclude that t_i and b_i are independent of each other and hence probing either Line 4 and 5 meets 1-PINI requirements. The probe on Line 7 would propagate to the registered values w_1 and w_2 revealing some information about t_2 and b_2 . As noted before, they are independent of each other and both are the output of two independent 1-PINI gadgets and hence meeting all requirements for 1-PINI. Consequently, Gadget 4 is first-order robust probing and 1-PINI secure. \square

Note that we have confirmed the 1-PINI security of Gadget 4 using the formal verification tool SILVER [KSM20].

4 Second-order Secure B2A Mask Conversion

In this section, we discuss second-order secure Boolean-to-Arithmetic mask conversion algorithms applied on integers over \mathbb{Z}_{2^k} (standard) and over a prime field with modulus q . We first revisit existing solutions and then make a proposal on how to implement such algorithms efficiently in hardware.

4.1 Hutter-Tunstall's Solution

The most efficient solution to perform a standard second-order secure Boolean-to-Arithmetic mask conversion is due to Hutter and Tunstall [HT16, HT19]. Similarly to Goubin's algorithm, they proposed a solution which is independent of the register size k and therefore has a complexity of $\mathcal{O}(1)$. Note that the solution was adopted and extended to higher-order security in [Cor17, BCZ18] but the explicit 2nd-order secure algorithm in [HT19] still remains the fastest solution with being 28% faster (and requiring only 5 instead of 7 random variables) than reported in [BCZ18].

Let's consider a Boolean masked input $x' = x \oplus r_1 \oplus r_2$, where $x, r_1, r_2 \in (\mathbb{Z}_{2^k}, \oplus, +)$, and an arithmetically masked output $x'' = x + s_1 + s_2$, where $s_1, s_2 \in (\mathbb{Z}_{2^k}, \oplus, +)$. Given x' , we want to compute x'' securely, i.e., without revealing any information about x in a side channel.

The secure mask conversion can then be performed in three steps:

1. We compute $(x + (r_1 \oplus r_2 \oplus \alpha)) + s_1$ for some random values $\alpha, s_1 \in \mathbb{Z}_{2^k}$.
2. We compute $s_2 - (r_1 \oplus r_2 \oplus \alpha)$ for some random $s_2 \in \mathbb{Z}_{2^k}$.
3. We add the results of Steps 1 and 2 together which results in $x'' = x + s_1 + s_2$.

Note that the algorithm can be executed using three first-order secure B2A mask conversions. Two are needed in Step 1 and one is needed in Step 2. The explicit instructions are provided in [Appendix A](#).

4.2 A New Second-Order Secure B2A_{2^k} Gadget Proposal

In the following, we propose a first and second-order secure sharing in hardware that fulfills the requirements of the PINI security model. We first discuss the sharing needed for Step 1 of the algorithm. The sharing for Step 2 is discussed afterwards. Finally, we combine the results in Step 3.

4.2.1 Step 1

Let's recall the computation in which we want to compute $(x + (r_1 \oplus r_2 \oplus \alpha)) + s_1$ for some random values $\alpha, s_1 \in \mathbb{Z}_{2^k}$. This can be done by first performing a secure B2A mask conversion to compute $(x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1$, followed by a second B2A mask conversion to produce $(x + (r_1 \oplus r_2 \oplus \alpha)) + s_1$. We now present a hardware gadget that performs those conversions in 4 clock cycles.

Cycle 1. Given the three input shares $(x' = x \oplus r_1 \oplus r_2, r_1, r_2)$, we can define the intermediate value a using three additional fresh random variables α, γ_1 , and γ_2 as follows:

$$\begin{aligned} a_1 &= x' & a_3 &= r_1 \oplus \gamma_1 & a_5 &= (x' \oplus \alpha \oplus \gamma_1) + \gamma_1 \\ a_2 &= \alpha & a_4 &= r_2 \oplus \gamma_2 & a_6 &= (x' \oplus \alpha \oplus \gamma_2) + \gamma_2 \end{aligned}$$

Cycle 2. We can define a set of shares b and use one output share s_1 that was chosen randomly from a uniform distribution, i.e.,

$$\begin{aligned} b_1 &= a_1 \oplus a_3 \oplus a_4 & b_3 &= s_1 \oplus a_5 \oplus a_3 & b_5 &= s_1 \\ b_2 &= a_2 \oplus a_3 \oplus a_4 & b_4 &= a_6 \oplus a_3 \end{aligned}$$

All shares represent terms that are needed for the first B2A mask conversion, i.e.,

$$\begin{aligned} (x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1 &= \Phi(x' \oplus \alpha, \gamma_1) \oplus \Phi(x' \oplus \alpha, \gamma_2) \oplus \\ &\Phi(x' \oplus \alpha, \gamma_1 \oplus \gamma_2 \oplus r_1 \oplus r_2 \oplus \alpha) \oplus s_1, \end{aligned} \quad (12)$$

where $b_1 + b_2 = \Phi(x' \oplus \alpha, \gamma_1 \oplus \gamma_2 \oplus r_1 \oplus r_2 \oplus \alpha)$, $a_5 = \Phi(x' \oplus \alpha, \gamma_1)$, and $a_6 = \Phi(x' \oplus \alpha, \gamma_2)$. Note that we need to add s_1 to share b_3 to avoid first-order leakage and also need to add a mask to a_5 and a_6 to avoid second-order leakage. We re-used variable a_3 for that purpose which does not further unmask the sensitive variable x .

Cycle 3. Now, we can define the set of shares c as follows:

$$\begin{aligned} c_1 &= (b_1 + b_2) \oplus b_3 & c_3 &= \delta & c_5 &= (b_5 \oplus \delta) + \beta \\ c_2 &= b_4 \oplus \delta & c_4 &= b_5 & c_6 &= \beta \end{aligned}$$

with

$$c_1 = \Phi(x' \oplus \alpha, \gamma_1) \oplus \Phi(x' \oplus \alpha, \gamma_1 \oplus \gamma_2 \oplus r_1 \oplus r_2 \oplus \alpha) \oplus s_1 \quad (13)$$

$$c_2 = \Phi(x' \oplus \alpha, \gamma_2) \oplus \delta \quad (14)$$

where $c_1 \oplus c_2 \oplus c_3 = (x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1$. Note that we add a fresh random variable δ to the term c_2 to avoid leakage in the subsequent B2A mask conversion applied in the next cycle.

Cycle 4. Now, we have all terms to compute $(x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1$ securely, we can apply another first-order secure B2A to obtain $(x + (r_1 \oplus r_2 \oplus \alpha)) + s_1$. We can define the set of shares d , i.e.,

$$\begin{aligned} d_1 &= (c_1 \oplus c_2 \oplus c_4) + c_5 & d_3 &= ((c_1 \oplus c_2) + c_3) \oplus c_2 & d_5 &= c_6 \\ d_2 &= c_1 \oplus c_3 & d_4 &= c_4 \end{aligned}$$

where $d_1 = \Phi((x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1, \delta \oplus s_1) + \beta$, $d_2 = (x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1 \oplus c_2$, and $d_3 = \Phi((x + (r_1 \oplus r_2 \oplus \alpha)) \oplus s_1, \delta) \oplus c_2$. Note that we needed to add a fresh random variable β to the term c_5 to avoid second-order leakage. This arithmetic mask gets then removed by a subtraction in the next cycle.

Output. The output of Step 1 can finally be computed by adding the shares d_2 , d_3 , and $(d_1 - d_5)$ together, i.e., $z_1 = d_2 \oplus d_3 \oplus (d_1 - d_5) = (x + (r_1 \oplus r_2 \oplus \alpha)) + s_1$.

4.2.2 Step 2

We want to compute $s_2'' = s_2 - (r_1 \oplus r_2 \oplus \alpha)$ given a Boolean masked input share $s_2' = s_2 \oplus r_1 \oplus r_2 \oplus \alpha$ securely using a B2A mask conversion. Let's consider the sharing m as follows:

$$\begin{aligned} m_1 &= s_2 \oplus r_1 & m_3 &= s_2 & m_5 &= \alpha \oplus r_2 \oplus s_2 \\ m_2 &= \alpha \oplus r_2 & m_4 &= r_1 \end{aligned}$$

then we can compute the second set of intermediate variables n , i.e.,

$$\begin{aligned} n_1 &= m_1 - m_2 & n_3 &= m_1 \oplus m_3 & n_5 &= m_5 - m_4 \\ n_2 &= m_2 \oplus m_3 & n_4 &= m_3 \end{aligned}$$

followed by

$$\begin{aligned} o_1 &= n_2 \oplus n_3 \oplus \delta & o_3 &= n_1 \\ o_2 &= n_5 \oplus \delta & o_4 &= n_4 \end{aligned}$$

where o_1 corresponds to $s_2' \oplus \delta$, o_2 corresponds to $\bar{\Phi}(s_2', r_1) \oplus \delta$, and o_3 corresponds to the term $\bar{\Phi}(s_2', r_2 \oplus \alpha)$, with $\bar{\Phi}$ performing a subtraction instead of an addition, i.e., a function that uses an addition with the additive inverse of an operand. We can define it as follows:

$$\begin{aligned} \bar{\Phi}(a, b) : (\mathbb{Z}_{2^k}, \oplus, +)^2 &\longrightarrow (\mathbb{Z}_{2^k}, \oplus, +) \\ a, b &\longmapsto (a \oplus b) - b \end{aligned} \tag{15}$$

for any $k \in \mathbb{Z}_{\geq 0}$. Using these three shares, one can then securely compute:

$$s_2'' = \bigoplus_{i=1}^3 o_i = o_1 \oplus o_2 \oplus o_3 = s_2 - (r_1 \oplus r_2 \oplus \alpha).$$

Note that a mask δ was added to o_1 and o_2 to avoid leakage when combining the three shares. We can re-use the same δ as used in Step 1 for that purpose without causing additional leakage. Also note that o_4 is not needed for calculating s_2'' but it is used as the final arithmetic output share z_3 in Step 3.

Gadget 5: 20-SecB2A2k: 2-PINI Secure B2A_{2k} Mask Conversion (pipelineable)**Input:** Shares $x' = x \oplus r_1 \oplus r_2$, the masks r_1 and r_2 , random integers $s_1, s_2, \gamma_1, \gamma_2, \alpha, \beta, \delta$, where all variables are $\in \mathbb{Z}_{2^k}$ **Output:** $z = (z_1, z_2, z_3)$ where $x = z_1 - z_2 - z_3$

Cycle 1:	17 $b_4 \leftarrow a_6 \oplus a_3$	33 $o_3 \leftarrow n_1$
2 $a_1 \leftarrow x'$	18 $b_5 \leftarrow s_1$	34 $o_4 \leftarrow n_4$
3 $a_2 \leftarrow \alpha$	19 $n_1 \leftarrow m_1 - m_2$	Cycle 4:
4 $a_3 \leftarrow r_1 \oplus \gamma_1$	20 $n_2 \leftarrow m_2 \oplus m_3$	36 $d_1 \leftarrow (c_1 \oplus c_2 \oplus c_4) + c_5$
5 $a_4 \leftarrow r_2 \oplus \gamma_2$	21 $n_3 \leftarrow m_1 \oplus m_3$	37 $d_2 \leftarrow c_1 \oplus c_3$
6 $a_5 \leftarrow (x' \oplus \alpha \oplus \gamma_1) + \gamma_1$	22 $n_4 \leftarrow m_3$	38 $d_3 \leftarrow ((c_1 \oplus c_2) + c_3) \oplus c_2$
7 $a_6 \leftarrow (x' \oplus \alpha \oplus \gamma_2) + \gamma_2$	23 $n_5 \leftarrow m_5 - m_4$	39 $d_4 \leftarrow c_4$
8 $m_1 \leftarrow s_2 \oplus r_1$	Cycle 3:	40 $d_5 \leftarrow c_6$
9 $m_2 \leftarrow \alpha \oplus r_2$	25 $c_1 \leftarrow (b_1 + b_2) \oplus b_3$	41 $p_1 \leftarrow o_1 \oplus o_2 \oplus o_3$
10 $m_3 \leftarrow s_2$	26 $c_2 \leftarrow b_4 \oplus \delta$	42 $p_2 \leftarrow o_4$
11 $m_4 \leftarrow r_1$	27 $c_3 \leftarrow \delta$	Outputs:
12 $m_5 \leftarrow \alpha \oplus r_2 \oplus s_2$	28 $c_4 \leftarrow b_5$	44 $z_1 \leftarrow d_2 \oplus d_3 \oplus (d_1 - d_5)$
14 $b_1 \leftarrow a_1 \oplus a_3 \oplus a_4$	29 $c_5 \leftarrow (b_5 \oplus \delta) + \beta$	45 $z_2 \leftarrow d_4 - p_1$
15 $b_2 \leftarrow a_2 \oplus a_3 \oplus a_4$	30 $c_6 \leftarrow \beta$	46 $z_3 \leftarrow p_2$
16 $b_3 \leftarrow s_1 \oplus a_5 \oplus a_3$	31 $o_1 \leftarrow n_2 \oplus n_3 \oplus \delta$	return $z = (z_1, z_2, z_3)$
	32 $o_2 \leftarrow n_5 \oplus \delta$	

4.2.3 Step 3

We can combine the results from Step 1 and Step 2 to obtain the arithmetic sharing z , i.e.,

$$\begin{aligned}
 z_1 &= d_2 \oplus d_3 \oplus (d_1 - d_5) = (x + (r_1 \oplus r_2 \oplus \alpha)) + s_1 \\
 z_2 &= d_4 - (o_1 \oplus o_2 \oplus o_3) = s_1 - s_2'' = s_1 - (s_2 - (r_1 \oplus r_2 \oplus \alpha)) \\
 z_3 &= o_4 = s_2
 \end{aligned}$$

where $x = z_1 - z_2 - z_3$.

Gadget 5 shows the hardware description of our solution. Note that this version is fully pipelinable and can accept new inputs in each clock cycle. The latency of the gadget is 4 clock cycles and it is therefore independent of the input word size k , i.e., it has a complexity of $\mathcal{O}(1)$. In total, the gadget needs 7 fresh random variables in total, e.g., 224 bits for a 32-bit B2A. The proposed gadget is second-order probing secure and composable in the PINI and SNI security models and was verified using SILVER. The Verilog code based on this algorithm is provided in Appendix C.

Theorem 4. *Gadget 5 is second-order glitch-extended PINI (2-PINI).*

Proof. All intermediate values in Cycle 1 are masked by a random value except those that are used to store an input share that is needed in other cycles, e.g., a_1 and m_4 . The gadget is designed in a way that probing any two intermediate values (internal probes) in Cycles 1 to 4 would leak information about only two share domains at most. Since p_1, p_2 , and d_1 to d_5 are all can be perfectly simulated with random inputs $\{s_1, s_2, \gamma_1, \gamma_2, \alpha, \beta, \delta\}$, we can conclude that any set of 2 probes on output shares (z_1, z_2, z_3) would leak no information about the input shares. As a result, Gadget 5 meets all requirements for 2-PINI security notion. \square

Besides the theoretical proof, we confirmed the second-order glitch-robust PINI security of Gadget 5 using the formal verification tool SILVER [KSM20].

Gadget 6: 20-KSABorrowBitGen: 2-PINI Secure Borrow-Bit Generator

Input: Shares $x = (x_0, x_1, x_2)$, where $x_0, x_1, x_2 \in \mathbb{Z}_{2^k}$, modulus $q < 2^k$.
Output: Borrow bit $b = (b_0, b_1, b_2)$ generated during $x - q$ where $b_0, b_1, b_2 \in \mathbb{F}_2$

```

1 for  $i := 0$  to  $k - 2$  by 2 do
2    $(a_0^i, a_1^i, a_2^i) \leftarrow \text{Refresh}(x_0^i, x_1^i, x_2^i)$ 
3 end
4 for  $i := 1$  to  $k - 1$  by 2 do
5    $(a_0^i, a_1^i, a_2^i) \leftarrow (x_0^i, x_1^i, x_2^i)$ 
6 end
7  $u \leftarrow 2^k - q$ 
8 for  $i := 1$  to  $k - 1$  by 2 do
9    $(p_0^i, p_1^i, p_2^i) \leftarrow \text{20-DomAnd}(a_0^i \oplus u^i, a_1^i, a_2^i, a_0^{i-1} \oplus u^{i-1}, a_1^{i-1}, a_2^{i-1})$ 
10   $(g_0^i, g_1^i, g_2^i) \leftarrow \text{20-DomAndRefresh}(a_0^i, a_1^i, a_2^i, a_0^{i-1}, a_1^{i-1}, a_2^{i-1}, u^i, u^{i-1})$ 
11 end
12 for  $j := 2$  to  $\log_2(k) - 1$  do
13    $\alpha \leftarrow k$ 
14    $\beta \leftarrow 2^{j-1}$ 
15   for  $i := 1$  to  $\frac{k}{2^j}$  do
16      $m \leftarrow \alpha - 1$ 
17      $l \leftarrow \alpha - \beta - 1$ 
18      $(t_0, t_1, t_2) \leftarrow (p_0^m, p_1^m, p_2^m)$ 
19      $(p_0^m, p_1^m, p_2^m) \leftarrow \text{20-DomAnd}(t_0, t_1, t_2, p_0^l, p_1^l, p_2^l)$ 
20      $(g_0^m, g_1^m, g_2^m) \leftarrow \text{20-DomAndXorRefresh}(t_0, t_1, t_2, g_0^l, g_1^l, g_2^l, g_0^m, g_1^m, g_2^m)$ 
21      $\alpha \leftarrow \alpha - 2\beta$ 
22   end
23 end
24  $m \leftarrow k - 1$ 
25  $l \leftarrow \frac{k}{2} - 1$ 
26  $(b_0, b_1, b_2) \leftarrow \text{20-DomAndXorRefresh}(p_0^m, p_1^m, p_2^m, g_0^l, g_1^l, g_2^l, g_0^m, g_1^m, g_2^m)$ 
27 return  $(b_0, b_1, b_2)$ 

```

4.3 A Novel Second-Order Secure B2A_q Gadget Proposal

We now present a second-order secure hardware gadget to perform a B2A mask conversion over integers modulo an arbitrary modulus $q < 2^k$. We consider the main idea described in Section 3.4 but replace all first-order secure with second-order secure gadgets. We first explain how we extend Gadget 3 (the borrow bit of $x - q$) to second-order security, which is outlined in Gadget 6. Since second-order security is desired, we need 3 shares to perform the computation. Similar to first-order gadget, we need to refresh all even input shares. To do so, the **Refresh** gadget takes the even bits of input shares (x_0, x_1, x_2) and generate the refreshed outputs $(x_0 \oplus r_0 \oplus r_1, x_1 \oplus r_1 \oplus r_2, x_2 \oplus r_2 \oplus r_0)$. The rest of the calculation is performed by employing second-order secure gadgets described in Subsection 2.3 all of which are executed in a single cycle. The randomness complexity of **20-DomAnd** is 3 fresh masks per gadget call. For **20-DomAndRefresh** and **20-DomAndXorRefresh**, the randomness requirement increases to 9 bits per gadget call. The latency of **20-KSABorrowBitGen** is also similar to **10-KSABorrowBitGen**, where one cycle is needed for refreshing the input shares and $\log_2(k)$ cycles to perform the calculations.

Gadget 7 shows the proposed algorithm for performing B2A_q. It operates with a complexity of $\mathcal{O}(\log k)$ and requires $\max\{5, \log_2(k) + 2\}$ clock cycles to execute B2A_q. The B2A_{2^k} operation necessitates at least 4 clock cycles, whereas multiplexing only requires 1 cycle. Thus, the minimum latency is 5 clock cycles. For $k > 8$, **20-KSABorrowBitGen** becomes the bottleneck, determining the latency as $\log_2(k) + 2$. The randomness requirement

Gadget 7: 20-SecB2Aq: 2-PINI Secure B2A_q Mask Conversion

Input: Shares $x' = x \oplus r_1 \oplus r_2$, r_1 and r_2 , where $x, r_1, r_2 \in \mathbb{Z}_{2^k}$, modulus $q < 2^k$.

Output: $z = (z_1, z_2, z_3)$ where $x = z_1 - z_2 - z_3$ with $x \in [0, q - 1]$

```

1  $(t_1, t_2, t_3) \leftarrow \text{20-SecB2A2k}(x', r_1, r_2)$ 
2  $u \leftarrow t_3 + q$ 
3  $v \leftarrow t_3 \oplus u$ 
4  $(b_1, b_2, b_3) \leftarrow \text{20-KSABorrowBitGen}(x', r_1, r_2, q)$ 
5  $\alpha \xleftarrow{\$} \mathbb{F}_2^{\frac{k}{2}}$ 
6  $w_1 \leftarrow (b_1 \wedge v) \oplus t_3$ 
7  $w_2 \leftarrow (b_2 \wedge v) \oplus \alpha$ 
8  $w_3 \leftarrow (b_3 \wedge v) \oplus \alpha$ 
9  $w_4 \leftarrow w_1 \oplus w_2$ 
10  $(y_1, y_2) \leftarrow \text{10-SecB2A2k}(w_4, w_3)$ 
11  $z_1 \leftarrow t_1 + y_2$ 
12  $z_2 \leftarrow t_2$ 
13  $z_3 \leftarrow y_1$ 
14 return  $z = (z_1, z_2, z_3)$ 

```

of the gadget is $7k$ bits per cycle for a k -bit input.

Theorem 6. *Gadget 7 is second-order glitch-extended PINI (2-PINI).*

Proof. We first discuss the 2-PINI security of the 20-KSABorrowBitGen gadget, where *refresh-then-multiply* technique is used similar to the first-order secure version. In Line 2 of Gadget 6, we refresh one of the inputs to the DOM AND gates. We use 20-DomAndRefresh to generate (g_0^i, g_1^i, g_2^i) while being refreshed for further use. Note that using 20-DomAndRefresh is not necessary to calculate (p_0^i, p_1^i, p_2^i) in Lines 9 and 19, as the refreshing in Line 2 is sufficient to satisfy 2-PINI requirements for the entire design similar to the first-order version, as one of the inputs to the 20-DomAnd gadget is already refreshed in Line 2. We confirmed the 2-PINI security of 20-KSABorrowBitGen using SILVER as well.

Since 20-SecB2A2k and 20-KSABorrowBitGen are 2-PINI secure, b_i and t_i are totally independent of each other. Examining Line 2, we observe that u carries the same information as t_3 since it results from an addition with a constant. Additionally, all values of w_i are random. Specifically, w_1 is randomized by t_3 , while w_2 and w_3 are randomized by the fresh mask α . Probing w_4 in Line 9 would reveal information about the registered values w_1 and w_2 . Hence, the extended probe on w_1 would reveal information about t_3 while the extended probe on w_2 would reveal information about α . Hence, probing any other intermediate values cannot violate 2-PINI notion. Particularly, probing w_3 and w_4 would reveal information about b_3 at most due to the use of the fresh mask α and hence is 2-PINI. In Line 10, we use our 1-PINI 10-SecB2A2k gadget to obtain arithmetic shares (y_1, y_2) where $y_2 - y_1 = w_1 \oplus w_2 \oplus w_3$. 10-SecB2A2k is 1-PINI and hence any probe in this gadget reveal information about either w_3 or w_4 by definition so to gain information about both w_3 and w_4 (or y_1 and y_2) the adversary needs to use two probes and as stated above it reveals information only about b_3 and hence is 2-PINI. y_i and t_i are totally independent of each other and having information about both y_1 and y_2 can only reveal b_3 . As a result, probing any set of two output shares cannot violate the 2-PINI security of the gadget. As a result, Gadget 7 meets all requirements for 2-PINI security notion. \square

In addition to the theoretical proof, we verified the second-order glitch-extended PINI security of Gadget 7 using the formal verification tool SILVER [KSM20].

Table 1: Performance comparison of 1st-order secure $B2A_{2^{32}}$ mask conversions.

Design	Type	Arch.	Lat. [cycles]	Rand [bits] ^a	FPGA Performance					Device
					Freq [MHz]	TP [Mbits/s]	Resources LUTs FFs DSPs			
[SMG15]	secADD/masked adder based	TI RCA	32	4	101	101	227	223	0	Spartan-6
		TI KSA	6	31	62	330	937	1,330	0	Spartan-6
[BG22]		HPC2 KSA	12	249	176	469	2,936	3,981	0	Spartan-6
		TI KSA	6	31	228	1,216	873	1,416	0	Spartan-6
[FBR ⁺ 22]		TI KSA	6	160 ^b	454	2,421	2,464	1,323	0	Artix-7
[NDKV24] ^c		HPC3 KSA	10	283	150	4,800	1,638	2,874	0	Kintex-7
Ours (F)	affine property		1	32	502	16,064	188	126	0	Kintex-7
Ours (S)					285	9,120	76	94	1	

^aThe column displays the random bit requirements per cycle.

^bThe random bits/cycle are estimated from the paper.

^cThe reported numbers in [NDKV24] are for $k = 13$, not $k = 32$. Therefore, a direct comparison is not feasible, and the performance is expected to be significantly lower than reported in this table.

5 Results and Comparison with Related Work

In this section, we present implementation results of our proposed hardware gadgets and compare it with related work. Performance numbers have been obtained from synthesis on a Kintex-7 XC7K160T FPGA using Vivado. Each gadget was embedded in a testbench that flopped the inputs and outputs. The maximum frequency was determined by increasing the frequency until occurrence of a timing violation. We didn't allow the compiler to flatten the hierarchy and applied a `dont_touch` directive on each gadget module boundary to prevent synthesis from optimization. To enforce use of DSPs, we applied the `use_dsp` directive, as additions and subtractions are implemented with logic instead of with DSP blocks per default.

5.1 $B2A_{2^k}$ Mask Conversion Performance

Table 1 shows the implementation results of our 10-SecB2A2k gadget and compares the results with related work. We first compare only solutions protecting against first-order attacks and consider $k = 32$ bits as used in previous works. To the best of our knowledge, all previous works propose to implement a masked adder chain to perform an addition over Boolean shares or to realize a secure B2A or A2B mask conversion.

One of the first proposals of such masked adders was made by Schneider, Moradi, and Güneysu [SMG15] in 2015. They proposed a Ripple-Carry Adder (RCA) and Kogge-Stone Adder (KSA) that is masked using a 3-share Threshold implementation. Their RCA-based solution needs 32 clock cycles and requires 227 LUTs and 223 FFs on a Spartan-6 FPGA. Their KSA-based design needs only 6 cycles but requires more resources, i.e., 937 LUTs and 1,330 FFs. Similar results were also reported by Bache et al. [BG22] in 2022. They evaluated different types of adder structures and applied a 2-share masking scheme based on the 1-PINI-secure HPC1 AND gadgets. Their TI-based KSA adder improves the work of [SMG15] in terms of LUT resource requirements and critical path. Their KSA-based adder needs 12 clock cycles and 2,936 LUTs and 3,981 FFs. In the same year, Fritzmann et al. [FBR⁺22] proposed a similar KSA-based solution and reported numbers on an Artix-7 FPGA.

Hardware synthesis numbers of Boolean masked adders were also reported by Cassiers et al. [CGM⁺23] and Knichel et al. [KMMS22]. They proposed and discussed the generation

Table 2: Performance comparison of 2nd-order secure B2A_{2³²} mask conversions.

Design	Type	Arch.	Lat. [cycles]	Rand [bits]	FPGA Performance					Device
					Freq [MHz]	TP [Mbits/s]	Resources			
						LUTs	FFs	DSPs		
[SMG15]	masked adder	TI RCA	65	8	107	52	388	387	0	Spartan-6
		TI KSA	12	128	63	168	4,223	5,509	0	
[BG22]		HPC2 KSA	12	747	148	395	3,915	8,001	0	Spartan-6
[NDKV24] ^a		HPC3 KSA	19	1,080	125	4,000	7,946	18,032	0	Kintex-7
Ours (F)	affine	pipelined	4	224	515	16,480	912	992	0	Kintex-7
Ours (S)	prop.				392	12,544	555	704	10	
Ours (F)	affine	iterative	4	128	434	3,472	968	333	0	
Ours (S)	prop.				322	2,576	900	330	2	

^aThe reported numbers in [NDKV24] are for $k = 13$, not $k = 32$. Therefore, a direct comparison is not feasible, and the performance is expected to be significantly lower than reported in this table.

of Boolean masked adder logic using automated tools such as AGEMA and COMPRESS. No explicit FPGA results were reported but their solution is also based on efficient masked adder structures (e.g., KSA-based) and have therefore similar performance expectations as previous works.

Recently, Norga et al. [NDKV24] presented a new B2A solution that is based on the SecAdd architecture proposed by [CGV14]. They improved performance by eliminating the pre-and post-processing stages needed in the SecAdd-based B2A mask conversion. It is important to note that they reported numbers for $k = 13$ only, which makes a fair comparison with other solutions using $k = 32$ impossible. However, we can observe that their 13-bit B2A solution has a latency requirement of 10 clock cycles and consumes 1,638 LUTs and 2,874 FFs.

As opposed to previous work, our 32-bit 10-SecB2A2k gadget needs only 1 clock cycle of latency and outperforms previous works by a factor of 6. It is not based on a masked-adder structure but exploits the affine property of Goubin’s solution to securely add the needed carries without the need of a masked carry chain. We report performance numbers of our gadget using two different target configurations: a *fast* (F) configuration that uses no DSPs and a *small* (S) configuration that enforces the compiler to use the available DSP48 slices. Note that one of the major advantages of using *non-masked* adders is that additions and subtractions, respectively, inside the gadget can be outsourced into DSPs, which is usually not possible using masked adders. Our results show that the use of DSPs reduces LUT and FF requirements but will also increase the net delay due to the longer wiring from and to the DSP units. In our experiments, the synthesizer outsourced the addition of r and γ (see Line 3 of Gadget 2) into a DSP48. Our *fast* configuration achieves a maximum frequency of 502 MHz and a throughput of 16 Gbits/s and is therefore 3 times faster than the fastest reported solution from [NDKV24]. Our *small* configuration can be clocked at 363 MHz but the lowest resource requirements were identified when clocking up to a frequency of 285 MHz where it requires only 76 LUTs, 94 FFs, and 1 DSP. It is therefore more than 2.5 times smaller than the smallest solution from [SMG15]. In addition, both configurations are more than a magnitude times smaller compared to all reported KSA-based designs.

Table 2 shows the performance numbers of our 20-SecB2A2k gadget and compares the result with other second-order secure implementations. For the comparison we again assume $k = 32$ bits. As opposed to related work, our solution needs a fixed latency of 4 cycles for any value of k . We implemented a *pipelined* and an *iterative* version of our gadget. The pipelined version is able to receive and output words in each clock cycle. The

Table 3: Performance results of our $B2A_q$ mask conversion gadgets.

Gadget Name	Sec. Order t	q^a [bits]	Latency [cycles]	Rand [bits/Cy]	FPGA Performance				
					Freq [MHz]	TP [Mbits/s]	Resources		
							LUTs ^b	FFs	DSPs
10-SecB2Aq (pipelined)	1	4	4	6	606	2,424	62 (47)	58	2
		8	5	12	588	4,704	132 (106)	126	2
		12 ^c	6	18	600	7,200	211 (166)	187	2
		16	6	24	555	8,880	298 (236)	262	2
		23 ^d	7	34	500	11,500	398 (310)	362	2
		32	7	48	454	14,528	653 (458)	536	2
20-SecB2Aq (pipelined)	2	4	7	28	434	1,736	179 (141)	192	13
		8	7	56	434	3,472	389 (310)	404	13
		12 ^c	8	84	408	4,896	597 (497)	607	13
		16	8	112	400	6,400	815 (659)	828	13
		23 ^d	9	155	370	8,510	1,079 (945)	1,176	13
		32	9	224	350	11,200	1,395 (1,307)	1,676	13

^a q is a k -bit prime number, with results reported for different values of k , as shown in the table.

^bNumbers in brackets show the LUT requirements when the gadget is clocked at 200 MHz.

^cWe used the 12-bit ML-KEM (Kyber) modulus, i.e., $q = 3329$.

^dWe used the 23-bit ML-DSA (Dilithium) modulus, i.e., $q = 8380417$.

iterative version can accept new inputs only every 4 clock cycles and re-uses the registers and adders to lower the resource requirements. In fact, 10 registers are needed in total which get re-used and overwritten in each cycle. As only 2 adders are required per cycle (see Gadget 5), we opted to recycle them and incorporated multiplexers accordingly. We further report performance numbers using a *fast* (F) and *small* (S) configuration.

Our pipelined solution achieves a throughput of 16.48 Gbits/s and needs 912 LUTs and 992 FFs. We can trade throughput for a lower resource requirement by using available DSPs. In this case, 10 DSPs are used (according to the number of total additions needed for 20-SecB2A2k). Our *small* configuration, in contrast, reduces the LUT requirement by 40% and FF requirements by 30%, i.e., 555 LUTs and 704 FFs. Our iterative version shows a significant reduction in FF requirements as expected. Only 333 FFs are needed, which is a reduction by 67% (without DSPs) and 52% (with DSPs) compared to the pipelined version. Inferring DSPs does not impact QoR by a lot in our experiments as there are only 2 adders needed in the construction. Still a LUT count reduction of 68 was achieved.

The presented numbers show that our gadgets outperform all related works that are based on masked Kogge-Stone adders. Compared to masked Ripple Carry Adders (RCA), our solution requires more resources than [SMG15] but reduces the latency requirements by a factor of 16. Our *iterative* solution can accept new inputs every 4 clock cycles and therefore offers a lower throughput of 3,472 Mbits/s. But it effectively trades throughput for lower register requirements: only half of fresh random bits are needed in each cycle and the number of flip flops got reduced by at least a factor of 2.

5.2 $B2A_q$ Mask Conversion Performance

Table 3 shows the implementation results of our 10-SecB2Aq and 20-SecB2Aq gadgets. It lists the performance data for word sizes $k = 4, 8, 12, 16, 23$, and 32. Please note that the numbers provided are for our *pipelinable* gadgets, indicating potential for further reductions in resource requirements if pipelining is not needed.

Our 10-SecB2Aq gadget has a latency of $\log_2(k) + 2$. The 10-KSABorrowBitGen gadget

Table 4: Performance comparison of pipelined $B2A_q$ mask conversions with modulus $q = 3329$ (ML-KEM), i.e., $k = 12$ bits., on the Kintex-7 XC7K160T.

Design	Security	Latency [cycles]	Rand [bits]	FPGA Performance				
	Order t			Freq [MHz]	TP [Mbits/s]	Resources		
						LUTs	FFs	DSPs
[NDKV24]	1	19	527	150	1,800	1,638	2,874	0
	2	37	2,056	125	1,500	7,946	18,032	0
10-SecB2Aq	1	6	18	600	7,200	211	187	2
20-SecB2Aq	2	8	84	408	4,896	597	607	13

requires $\log_2(k) + 1$ cycles and 1 cycle is needed for the secure multiplexing. Our gadget needs $3k/2$ bits of randomness in each cycle. For $k = 32$, 10-SecB2Aq needs 653 LUTs and 536 FFs when clocked at the maximum frequency of 454 MHz. Note that 2 DSPs can be used if available: one to outsource the addition of $r + \gamma$ inside the 10-SecB2A2k gadget, and one to outsource the addition of $t_2 + q$ in Line 2 of Gadget 4. We also synthesized the gadget at a lower frequency of 200 MHz and report the LUT count reduction in brackets. At this frequency the 10-SecB2Aq gadget needs only 458 LUTs and 536 FFs, where 276 LUTs and 312 FFs are consumed by the 10-KSABorrowBitGen gadget and the remaining 182 LUTs and 224 FFs are due to 10-SecB2A2k and the logic for the masked multiplexer.

Our 20-SecB2Aq gadget has a latency requirement of $\max\{7, \log_2(k) + 4\}$. At least 4 clock cycles are required for the B2A operation, while 3 cycles are necessary for the multiplexing. For $k > 8$, the 20-KSABorrowBitGen gadget becomes the bottleneck. For $k = 32$, our gadget needs 1,395 LUTs and 1,676 FFs when clocked at a maximum frequency of 350 MHz. 13 DSPs were inferred in our experiments: 10 for all the additions/subtractions inside the 20-SecB2A2k gadget, 1 for the addition of $t_3 + q$ (see Line 2 in Gadget 7), one addition inside the 10-SecB2A2k gadget, and one addition to generate the output share z_1 . Note that when DSPs are not inferred, this gadget needs 1,885 LUTs and 2,160 FFs and achieves a maximum frequency of 500 MHz. Compared to masked Kogge-Stone adders, this is an improvement by a factor of ≥ 3 in terms of resource and latency requirements.

Finally, we compare our results with the work of [NDKV24], who report $B2A_q$ performance numbers for the Kyber (ML-KEM) modulus $q = 3329$, i.e., $k = 12$ bits. Their first-order solution needs 19 clock cycles and 527 bits of randomness. Our solution needs 6 clock cycles and 18 random bits. This is more than 3 times faster in terms of clock cycles, and more than a magnitude times better in terms of randomness requirements. Our achieved throughput is 4 times higher than reported by [NDKV24]. In terms of resource requirements, their first-order solution needs 1,638 LUTs and 2,874 FFs. Our solution needs 211 LUTs, 187 FFs, and 2 DSPs. This is a LUT count reduction by about 87%, and a FF count reduction by about 93%. Their second-order solution needs 7,946 LUTs and 18,032 FFs, whereas our solution needs only 597 LUTs, 607 FFs, and 13 DSPs. This is a LUT count reduction by about 92%, and a FF count reduction by about 97%.

6 Practical Security Analysis

In this section, we present the practical results of our security analysis for our first- and second-order B2A mask conversion gadgets. We conducted a series of tests to evaluate their resilience against side-channel attacks, ensuring comprehensive validation of our security claims.

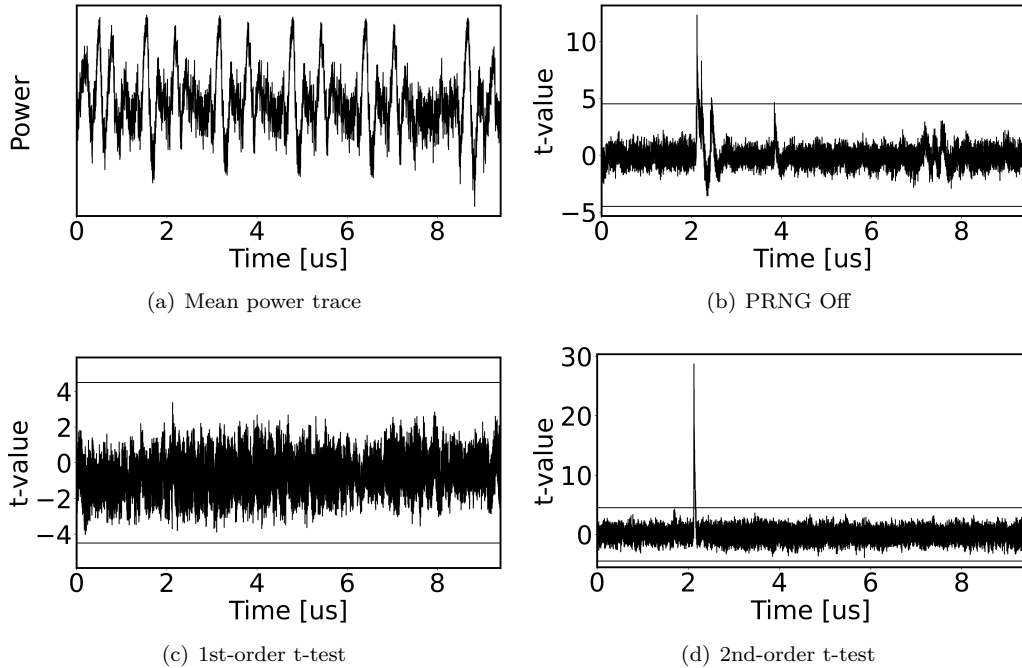


Figure 2: TVLA results of our 10-SecB2A2k gadget with 10 million traces.

6.1 Setup

We implemented our gadgets expressed in Section 3 and Section 4 on the target Xilinx Artix-7 FPGA of the CW305 NewAE board [New]. We collected power consumption traces by monitoring the voltage drop over a shunt resistor placed in the V_{dd} path of the target FPGA using a digital storage oscilloscope (Picoscope 6424E) at a sampling rate of 1.25 GS/s and a bandwidth of 500 MHz. During the measurements, the target FPGA were supplied by a stable clock source at the frequency of 50 MHz. The target FPGA receives masked input (plaintext) and issues output (ciphertext) also in the same sharing form. The fresh mask bits are generated in the PC and sent over to the board. We provided a trigger signal for the oscilloscope to indicate the start and end of a B2A mask conversion operation.

6.2 TVLA Results

We evaluated our implementations using *non-specific fixed-vs-random* t-tests [GJJR11]. In these tests, two sets of data are collected: one with fixed input values and the other with random input values. The power traces from these sets are then analyzed to calculate the t-score, which measures the difference between the means of the two sets normalized by their variance, which is also known as a first-order univariate t-test. If the t-statistic exceeds a certain threshold, typically ± 4.5 sigma, it indicates a significant difference between the fixed and random sets (a confidence of roughly 99.999%), suggesting potential leakage of sensitive information. This test is widely used because it does not require detailed knowledge of the implementation or the specific leakage model, making it a general-purpose tool for evaluating the security of cryptographic devices.

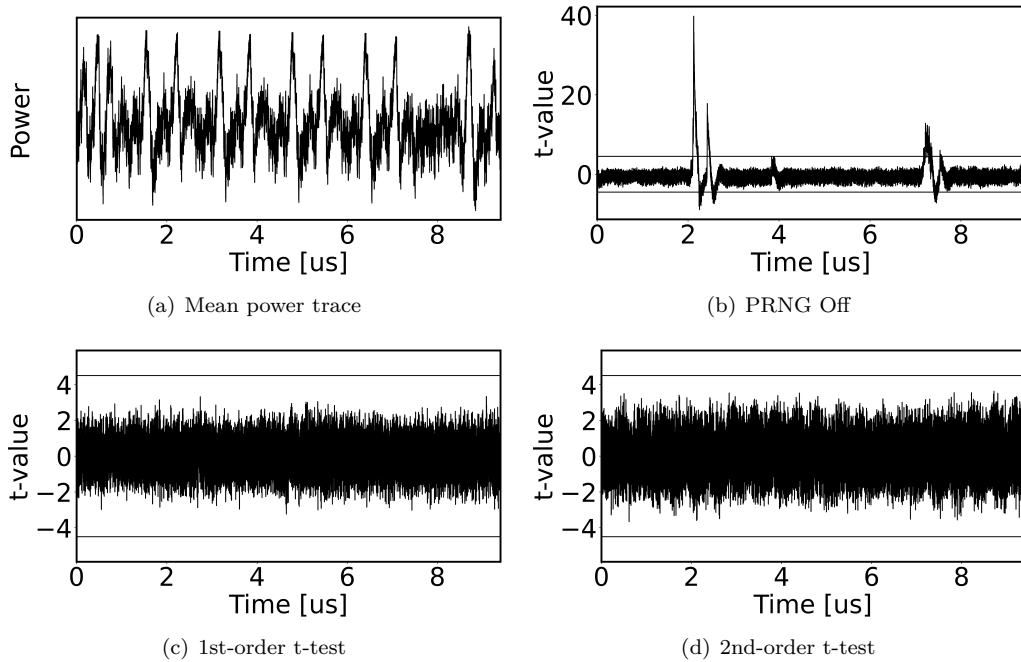


Figure 3: TVLA results of our 20-SecB2A2k gadget with 10 million traces.

6.2.1 1st Order T-test Result of 10-SecB2A2k

We first analyzed the security of our 10-SecB2A2k gadget that is running on our target FPGA. We decided to test a 16-bit, i.e., $k = 16$ mask conversion operation and stored the two input shares together with 2 additional random values (masks) in one 64-bit register. The mean power trace is shown in Figure 2(a). For this gadget, we collected 100,000 traces following the strategy explained in [GJJR11] to conduct a reliable fixed-versus-random t-test. We performed the ordinary t-test on each sample point individually to compute first-order t-test values, with setting the PRNG off (meaning setting all mask bits to zero) to demonstrate the setup’s ability to detect such leakage, with the corresponding result shown in Figure 2(b). There is a significant peak around μ s, which corresponds to the actual execution time of the mask-conversion process. We then enabled the countermeasure and collected 10 million power traces. We then performed the same test, where corresponding result shown in Figure 2(c) confirming its first-order security as expected.

We also performed a second-order t-test to confirm leakage. For this, we followed the techniques presented in [SM15] to efficiently compute the higher-order t-statistics. In particular, we made the traces mean-free for each group of fixed and random individually. Each mean-free sample point was then squared before calculating the t-statistics for univariate second-order t-tests. The corresponding result, shown in Figure 2(d), exhibiting a clear second-order leakage as anticipated.

We also performed the same tests using our 10-SecB2Aq gadget and couldn’t identify any significant leakages. We decided to omit those results in this paper to maintain conciseness.

6.2.2 2nd Order T-test Result of 20-SecB2A2k

We now report the t-test results of our second-order secure 20-SecB2A2k gadget. The mean power consumption trace is depicted in Figure 3(a). Note that we can observe a slightly higher mean power consumption at around 2.2μ s which is due to the fact that the

second-order secure gadget consists of more logical gates and consumes more power than the first-order secure gadget. The rest of the trace looks very similar as in Figure 2(a) because we used the same hardware wrapper to test both gadgets and the wrapper waits for a fixed period of 8 clock cycles to finish a mask conversion operation.

Similar to our previous experiment on 10-SecB2A2k, we first disabled the countermeasure and used 100,000 traces to prove that our setup is working as expected. Figure 3(b) shows the t-score trace which shows significant leakage at multiple points in time. The maximum t-value reaches almost 40 sigma. As a next step, we collected 10 million traces with countermeasure being enabled and performed first- and second-order T-tests to evaluate the security of our gadget. Figure 3(c) shows the 1st-order results and Figure 3(d) shows the 2nd-order results. As expected, no leakage was identified throughout the mask conversion operations.

Finally, we conducted the same tests using our 20-SecB2Aq gadget and, as expected, confirmed that there was no significant leakage. The detailed results are omitted to keep this section concise.

6.2.3 Summary of Practical Analysis

Our tests included the application of both first-order and second-order TVLA to our hardware gadgets, designed for secure Boolean-to-Arithmetic (B2A) mask conversion operations. The results from these tests validate the robustness and security of our gadgets under the tested conditions. This supports our claims of the gadgets' security and demonstrates their ability to resist side-channel attacks in practical scenarios. It also supports the claims made from our formal proofs that our gadgets are PINI secure. Therefore, we demonstrate the security of our gadgets not only from a theoretical standpoint but also through practical validation.

7 Conclusions

In this work, we revisited existing hardware implementation approaches to perform Boolean-to-Arithmetic (B2A) mask conversion efficiently. B2A is needed in many cryptographic systems including ARX-based designs, hash functions (e.g., SHA-2), Arithmetic Logic Units (e.g., in CPUs), or lattice-based cryptography (e.g., in Kyber or Dilithium). Previous work propose to use Boolean masked adders because they can be used to either perform B2A and A2B using a single hardware instance. However, those solutions suffer from 1) a long latency due to the masked adder chain delay, 2) a high resource requirement usually consuming > 1000 LUTs for a single 32-bit mask conversion, and 3) have the disadvantage that they are inefficient in case only B2A operations are needed or in cases when B2A and A2B need to be performed in parallel, e.g., in low-latency applications. Due to these reasons, we proposed an alternative solution that does not rely on Boolean masked adders. First, we presented a novel low-latency gadget that performs a $B2A_{2^k}$ in a single cycle. Then, we proposed a (pipelined) second-order secure $B2A_{2^k}$ that has a latency of 4 clock cycles and is independent of the input word size k . We also first present novel gadgets that perform $B2A_q$ mask conversion efficiently in hardware. Overall, our results show that our gadgets are more than a magnitude times smaller on a Kintex-7 FPGA compared to previously reported fast adder-based designs. All gadgets have been proven secure in the robust PINI model, and we have conducted practical TVLA tests to confirm our security claims.

Acknowledgments. The authors would like to thank Niels Samwel for helping out with the practical DPA testing and Graeme Hickey for his support throughout the project.

References

- [AHMP10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 Proposal BLAKE, December 2010. <https://131002.net/blake>.
- [BBD⁺15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified Proofs of Higher-Order Masking. In *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 457–485, Sofia, Bulgaria, April 2015. Springer, Heidelberg. doi:10.1007/978-3-662-46800-5_18.
- [BBD⁺16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong Non-Interference and Type-Directed Higher-Order Masking. In *CCS 2016*, pages 116–129, Vienna, Austria, October 2016. ACM. doi:10.1145/2976749.2978427.
- [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 *EUROCRYPT 2018*, volume 10821 of *LNCS*, pages 354–384, Tel Aviv, Israel, April 2018. Springer, Cham. doi:10.1007/978-3-319-78375-8_12.
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(4):553–588, 2022. doi:10.46586/TCHES.V2022.I4.553-588.
- [BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved High-Order Conversion From Boolean to Arithmetic Masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(2):22–45, 2018. doi:10.13154/TCHES.V2018.I2.22-45.
- [BDCU17] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal First-Order Boolean Masking for Embedded IoT Devices. In *CARDIS 2017*, volume 10728 of *LNCS*, pages 22–41, Lugano, Switzerland, November 2017. Springer. doi:10.1007/978-3-319-75208-2_2.
- [BDH⁺21] Shivam Bhasin, Jan-Pieter D’Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(3):334–359, 2021. doi:10.46586/TCHES.V2021.I3.334-359.
- [BDK⁺21] Michiel Van Beirendonck, Jan-Pieter D’Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A Side-Channel-Resistant Implementation of SABER. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(2):1–26, 2021. doi:10.1145/3429983.
- [Ber08] Daniel J. Bernstein. ChaCha, a Variant of Salsa20. In *State of the Art of Stream Ciphers Workshop (SASC)*, pages 3–5, Lausanne, Switzerland, Februar 2008. URL: <https://cr.yp.to/chacha.html>.
- [BFG⁺17] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, Clara Paglialonga, and François-Xavier Standaert. Consolidating Inner Product Masking. In *ASIACRYPT 2017*, volume 10624 of *LNCS*, pages 724–754, Hong Kong, China, December 2017. Springer. doi:10.1007/978-3-319-70694-8_25.

- [BG22] Florian Bache and Tim Güneysu. Boolean Masking for Arithmetic Additions at Arbitrary Order in Hardware. *Applied Sciences*, 12(5), 2022. URL: <https://doi.org/10.3390/app12052274>.
- [BPO⁺20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-Speed Masking for Polynomial Comparison in Lattice-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(3):483–507, 2020. doi:10.13154/TCHES.V2020.I3.483–507.
- [Cas22] Gaëtan Cassiers. *Composable and Efficient Masking Schemes for Side-channel Secure Implementations*. PhD thesis, Université Catholique de Louvain, June 2022. URL: <http://hdl.handle.net/2078.1/264203>.
- [CG00] Jean-Sébastien Coron and Louis Goubin. On Boolean and Arithmetic Masking against Differential Power Analysis. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 1965 of *LNCS*, pages 231–237, Worcester, MA, USA, August 2000. Springer, Heidelberg. doi:10.1007/3-540-44499-8_18.
- [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. IACR Cryptology ePrint Archive, Paper 2023/1600, 2023. URL: <https://eprint.iacr.org/2023/1600>.
- [CGMZ23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order Polynomial Comparison and Masking Lattice-based Encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(1):153–192, 2023. doi:10.46586/TCHES.V2023.I1.153–192.
- [CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In *FSE 2015*, volume 8731 of *LNCS*, pages 130–149, Istanbul, Turkey, March 2015. Springer, Heidelberg. doi:10.1007/978-3-662-48116-5_7.
- [CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved Gadgets for the High-Order Masking of Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(4):110–145, 2023. doi:10.46586/TCHES.V2023.I4.110–145.
- [CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 8731 of *LNCS*, pages 188–205, Busan, South Korea, September 2014. Springer, Heidelberg. doi:10.1007/978-3-662-44709-3_11.
- [Cor17] Jean-Sébastien Coron. Higher-Order Conversion from Boolean to Arithmetic Masking. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 10529 of *LNCS*, pages 93–114, Taipei, Taiwan, September 2017. Springer. doi:10.1007/978-3-319-66787-4_5.

- [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.
- [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 (TCHES)*, 2022(1):414–460, 2022. doi:10.46586/TCHES.V2022.I1.414-460.
- [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 (TCHES)*, 2018(3):89–120, 2018. doi:10.13154/TCHES.V2018.I3.89-120.
- [FLS⁺10] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family, October 2010. <https://www.schneier.com/wp-content/uploads/2015/01/skein.pdf>.
- [GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A Testing Methodology for Side-Channel Resistance Validation. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf, 2011. Non-Invasive Attack Testing (NIAT) Workshop.
- [GJM⁺16] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. Concealing Secrets in Embedded Processors Designs. In *CARDIS 2016*, volume 10146 of *LNCS*, pages 89–104, Cannes, France, November 2016. Springer. doi:10.1007/978-3-319-54669-8_6.
- [GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Workshop on Theory of Implementation Security (TIS)*, Vienna, Austria, October 2016. ACM. doi:10.1145/2996366.2996426.
- [Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 3–15, Paris, France, May 2001. Springer, Heidelberg. doi:10.1007/3-540-44709-1_2.
- [GP99] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 158–172, Worcester, MA, USA, August 1999. Springer. doi:10.1007/3-540-48059-5_15.
- [GT02] Jovan Dj. Golic and Christophe Tymen. Multiplicative Masking and Power Analysis of AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 198–212, San Francisco, USA, August 2002. Springer. doi:10.1007/3-540-36400-5_16.
- [HT16] Michael Hutter and Michael Tunstall. Constant Time Higher-Order Boolean-to-Arithmetic Masking. *IACR Cryptology ePrint Archive*, Paper 2016/1023, 2016. <https://eprint.iacr.org/2016/1023>.

- [HT19] Michael Hutter and Michael Tunstall. Constant-Time Higher-Order Boolean-to-Arithmetic Masking. *Journal of Cryptographic Engineering*, 9(2):173–184, 2019. doi:10.1007/S13389-018-0191-Z.
- [ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481, Santa Barbara, California, USA, August 2003. Springer, Heidelberg. doi:10.1007/978-3-540-45146-4_27.
- [KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397, Santa Barbara, California, USA, August 1999. Springer, Heidelberg. doi:10.1007/3-540-48405-1_25.
- [KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated Generation of Masked Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(1):589–629, 2022. doi:10.46586/TCHES.V2022.I1.589-629.
- [Kni23] David Knichel. *Formal Verification and Automated Masking of Cryptographic Hardware*. PhD thesis, Ruhr University Bochum, Germany, September 2023. URL: <https://hss-opus.ub.ruhr-uni-bochum.de/opus4/frontdoor/index/index/docId/10666>.
- [Koc96] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO 1996*, volume 1109 of *LNCS*, pages 104–113, Santa Barbara, California, USA, August 1996. Springer, Heidelberg. doi:10.1007/3-540-68697-5_9.
- [KS73] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, 22(8):786–793, 1973. doi:10.1109/TC.1973.5009159.
- [KSM19] Yuichi Komano, Hideo Shimizu, and Hideyuki Miyake. Integrative Acceleration of First-order Boolean Masking for Embedded IoT Devices. *Journal of Information Processing*, 27:585–592, 2019. doi:10.2197/IPSJJIP.27.585.
- [KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In *ASIACRYPT 2020*, volume 12491 of *LNCS*, pages 787–816, Daejeon, South Korea, December 2020. Springer. doi:10.1007/978-3-030-64837-4_26.
- [MBR19] Lauren De Meyer, Begül Bilgin, and Oscar Reparaz. Consolidating Security Notions in Hardware Masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(3):119–147, 2019. doi:10.13154/TCHES.V2019.I3.119-147.
- [MME10] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-Enhanced Power Analysis Collision Attack. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 6225 of *LNCS*, pages 125–139, Santa Barbara, California, USA, August 2010. Springer. doi:10.1007/978-3-642-15031-9_9.
- [MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully Attacking Masked AES Hardware Implementations. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 3659 of *LNCS*, pages 157–171, Edinburgh, Scotland, August 2005. Springer. doi:10.1007/11545262_12.

- [NDKV24] Quinten Norga, Jan-Pieter D’Anvers, Suparna Kundu, and Ingrid Verbauwhede. Mask Conversions for $d+1$ Shares in Hardware, with Application to Lattice-based PQC. IACR Cryptology ePrint Archive, Paper 2024/114/20240126:091208, 2024. <https://eprint.iacr.org/archive/2024/114/20240126:091208>.
- [New] NewAE. CW305 Artix FPGA Target. <https://rtfm.newae.com/Targets>.
- [NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *Information and Communications Security (ICICS)*, volume 4307 of *LNCS*, pages 529–545, Raleigh, NC, USA, December 2006. Springer, Heidelberg. doi:10.1007/11935308_38.
- [NW97] Roger M. Needham and David J. Wheeler. TEA Extensions. <https://www.cix.co.uk/~klockstone/xtea.pdf>, 1997. Technical Report, Computer Laboratory, University of Cambridge.
- [RBN⁺15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, volume 9215 of *LNCS*, pages 764–783, Santa Barbara, California, USA, August 2015. Springer. doi:10.1007/978-3-662-47989-6_37.
- [SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 9293 of *LNCS*, pages 495–513, Saint Malo, France, September 2015. Springer. doi:10.1007/978-3-662-48324-4_25.
- [SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic Addition Over Boolean Masking—Towards First- and Second-Order Resistance in Hardware. In *ACNS 2015*, volume 9092 of *LNCS*, pages 559–578, St.Petersburg, Russia, June 2015. Springer, Heidelberg. doi:10.1007/978-3-319-28166-7_27.
- [SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto. In *PKC 2019*, volume 11443 of *LNCS*, pages 534–564, Beijing, China, April 2019. Springer. doi:10.1007/978-3-030-17259-6_18.
- [WH17] Yoo-Seung Won and Dong-Guk Han. Efficient Conversion Method from Arithmetic to Boolean Masking in Constrained Devices. In *COSADE 2017*, volume 10348 of *LNCS*, pages 120–137, Paris, France, April 2017. Springer. doi:10.1007/978-3-319-64647-3_8.

A 2nd-order Secure $B2A_{2^k}$ Mask Conversion

Algorithm 4 shows the 2-SNI secure Boolean-to-arithmetic mask conversion algorithm proposed by Hutter and Tunstall [HT16, HT19]. It requires 31 instructions and 5 random values $(\gamma_1, \gamma_2, \alpha, s_1, s_2)$ to perform a mask conversion. The algorithm is independent of the input word size k .

Algorithm 4: Hutter-Tunstall's 2nd-Order Secure $B2A_{2^k}$ Mask Conversion.

Input: $x' = x \oplus r_1 \oplus r_2$ with $x, r_1, r_2 \in \mathbb{Z}_{2^k}$ and random numbers $\gamma_1, \gamma_2, \alpha, s_1, s_2 \in \mathbb{Z}_{2^k}$
for some $k \in \mathbb{Z}_{\geq 0}$

Output: $x'' = x + s_1 + s_2$

1 $z \leftarrow \gamma_1 \oplus r_1$	12 $w \leftarrow w + \gamma_2$	23 $w \leftarrow \alpha \oplus r_2$
2 $z \leftarrow z \oplus \gamma_2$	13 $z \leftarrow r_2 \oplus s_1$	24 $u \leftarrow s_2 \oplus r_1$
3 $z \leftarrow z \oplus r_2$	14 $u \leftarrow u \oplus r_2$	25 $u \leftarrow u - w$
4 $u \leftarrow x' \oplus z$	15 $u \leftarrow u \oplus v$	26 $w \leftarrow w \oplus s_2$
5 $z \leftarrow z \oplus \alpha$	16 $u \leftarrow u \oplus w$	27 $v \leftarrow w \oplus r_1$
6 $u \leftarrow u + z$	17 $v \leftarrow u \oplus s_1$	28 $w \leftarrow w - r_1$
7 $v \leftarrow x' \oplus \gamma_1$	18 $v \leftarrow v + r_2$	29 $u \leftarrow u \oplus v$
8 $v \leftarrow v \oplus \alpha$	19 $w \leftarrow u \oplus z$	30 $u \leftarrow u \oplus w$
9 $v \leftarrow v + \gamma_1$	20 $v \leftarrow v \oplus w$	31 $x'' \leftarrow z + u$
10 $w \leftarrow x' \oplus \gamma_2$	21 $w \leftarrow u + z$	return x''
11 $w \leftarrow w \oplus \alpha$	22 $z \leftarrow v \oplus w$	

B 1st-order PINI Secure $B2A_{2^k}$ Mask Conversion

Listing 1 shows the HDL code for a k -bit 1-PINI secure $B2A_{2^k}$ mask conversion based on Gadget 2, which is performed in a single cycle using a k -bit fresh mask g .

Listing 1: 1st-order PINI Secure $B2A_{2^k}$ Mask Conversion

```

1  module gadget_1o_b2a #(parameter K) (
2      input clk,
3      input [K-1:0] xp,      // input share 1: x' = x XOR r
4      input [K-1:0] r,      // input share 2: r
5      input [K-1:0] g,      // random mask (gamma)
6      output [K-1:0] z1,    // output share 1
7      output [K-1:0] z2    // output share 2
8  );
9
10     reg [K-1:0] a1, a2, a3, a4, z1;
11     reg T;
12     integer i, j, m, d;
13
14     always @(posedge clk) begin
15         a1 <= xp;
16         a2 <= xp ^ g;
17         a3 <= r + g;
18         a4 <= r ^ g;
19     end
20
21     always @(*) begin
22         z1[0] = a3[0];
23         z1[1] = a3[1] ^ (a1[0] & ~a4[0]);
24         for(i=2; i<K; i=i+1) begin
25             z1[i] = a3[i] ^ (a1[i-1] & (~a4[i-1]));
26             for (d=3; d<=i+1; d=d+1) begin
27                 j = i+1-d;
28                 t = a1[j] & (~a4[j]);
29                 for (m=d-2; m>=1; m=m-1) begin
30                     t = t & a4[j+m];
31                 end
32                 z1[i] = z1[i] ^ t;
33             end
34         end
35     end
36
37     assign z2 = a2;
38 endmodule

```

C 2nd-order PINI Secure $B2A_{2^k}$ Mask Conversion

Listing 2 shows the HDL code for a k -bit 2-PINI secure $B2A_{2^k}$ mask conversion based on Gadget 5, which has a latency of 4 clock cycles.

Listing 2: 2nd-order PINI Secure $B2A_{2^k}$ Mask Conversion

```

1  module gadget_2o_b2a #(parameter K) (
2      input clk,
3      input [K-1:0] xp,          // input share 1: x' = x XOR r1 XOR r2
4      input [K-1:0] r1,          // input share 2: r1
5      input [K-1:0] r2,          // input share 3: r2
6      input [K-1:0] a,           // random mask
7      input [K-1:0] g1,          // random mask
8      input [K-1:0] g2,          // random mask
9      input [K-1:0] s1,          // random mask
10     input [K-1:0] s2,          // random mask
11     input [K-1:0] d,           // random mask
12     input [K-1:0] n,           // random mask
13     output reg [K-1:0] z1,     // output share 1
14     output reg [K-1:0] z2,     // output share 2
15     output reg [K-1:0] z3     // output share 3
16 );
17
18     reg [K-1:0] a1, a2, a3, a4, a5, a6, m1, m2, m3, m4, m5;
19     reg [K-1:0] b1, b2, b3, b4, b5, n1, n2, n3, n4, n5;
20     reg [K-1:0] c1, c2, c3, c4, c5, c6, o1, o2, o3, o4;
21     reg [K-1:0] d1, d2, d3, d4, d5, p1, p2, z1, z2, z3;
22
23     // Cycle 1
24     always @(posedge clk) begin
25         a1 <= xp;
26         a2 <= a;
27         a3 <= r1 ^ g1;
28         a4 <= r2 ^ g2;
29         a5 <= (xp ^ a ^ g1) + g1;
30         a6 <= (xp ^ a ^ g2) + g2;
31
32         m1 <= s2 ^ r1;
33         m2 <= a ^ r2;
34         m3 <= s2;
35         m4 <= r1;
36         m5 <= a ^ r2 ^ s2;
37     end
38     // Cycle 2
39     always @(posedge clk) begin
40         b1 <= a1 ^ a3 ^ a4;
41         b2 <= a2 ^ a3 ^ a4;
42         b3 <= s1 ^ a5 ^ a3;
43         b4 <= a6 ^ a3;
44         b5 <= s1;
45
46         n1 <= m1 - m2;
47         n2 <= m2 ^ m3;
48         n3 <= m1 ^ m3;
49         n4 <= m3;
50         n5 <= m5 - m4;

```

```
51     end
52     // Cycle 3
53     always @(posedge clk) begin
54         c1 <= (b1 + b2) ^ b3;
55         c2 <= b4 ^ d;
56         c3 <= d;
57         c4 <= b5;
58         c5 <= (b5 ^ d) + n;
59         c6 <= n;
60
61         o1 <= (n2 ^ n3) ^ d;
62         o2 <= (n2 - n3) ^ d;
63         o3 <= n1;
64         o4 <= n4;
65     end
66     // Cycle 4
67     always @(posedge clk) begin
68         d1 <= (c1 ^ c2 ^ c4) + c5;
69         d2 <= c1 ^ c3;
70         d3 <= ((c1 ^ c2) + c3) ^ c2;
71         d4 <= c4;
72         d5 <= c6;
73
74         p1 <= o1 ^ o2 ^ o3;
75         p2 <= o4;
76     end
77     // Output
78     always @(*) begin
79         z1 = d2 ^ d3 ^ (d1 - d5); //x + (r1 ^ r2 ^ a) + s1
80         z2 = d4 - p1;           //s1 - (s2 - (r1 ^ r2 ^ a))
81         z3 = p2;               //s2
82     end
83 endmodule
```
