Evaluating Ascon in Secure Multi-Party Computation using Reverse Multiplication-Friendly Embeddings

Peter Schwarz
COSIC, KU Leuven
Leuven, Belgium
peter.schwarz@esat.kuleuven.be

Erik Pohle
COSIC, KU Leuven
Leuven, Belgium
erik.pohle@esat.kuleuven.be

Aysajan Abidin
COSIC, KU Leuven
Leuven, Belgium
aysajan.abidin@esat.kuleuven.be

Bart Preneel
COSIC, KU Leuven
Leuven, Belgium
bart.preneel@esat.kuleuven.be

Abstract

We present the first systematic study on communication-efficient evaluation of the lightweight cipher family Ascon within secure multi-party computation (MPC). By leveraging Ascon's parallel, bit-oriented structure, we adapt its design using Reverse Multiplication-Friendly Embeddings (RMFEs, introduced by Cascudo et al. in CRYPTO'18) in a single-circuit evaluation, enabling efficient packing of groups of bits into field elements.

Our protocol, which uses relatively small RMFEs, achieves substantial reductions in communication cost compared to baseline MPC protocols. For example, in a medium-sized setting (with $n = 13$ MPC parties), our protocol reduces the communication cost for an Ascon permutation by roughly 38%. For large amounts of parties (e.g., $n = 255$), the reduction can reach 50%. These improvements are achieved even though RMFEs only pack a few bits per field element, due to favorable amortization of both substitution and linear layers. We also provide a Boolean circuit implementation of Ascon in the MP-SPDZ framework, enabling straightforward benchmarking.

Our findings are particularly beneficial for bandwidth-constrained environments where the use of lightweight ciphers, such as Ascon, is necessary due to the resource limitations of client devices, as in the case of transciphering data from IoT sensors. Since our optimizations target the Ascon permutation, they naturally extend to all cryptographic modes (encryption, decryption, hashing) defined for the standard.

Keywords

Ascon, Secure Multi-Party Computation, Reverse Multiplication-Friendly Embeddings

1 Introduction

Secure multiparty computation (MPC, [27, 31]) enables n ($n \geq 2$) parties to jointly compute a function on their private inputs without revealing individual input values to any participant. This privacy guarantee has established MPC as a widely studied and versatile cryptographic tool. Different lines of work in MPC research focus on varying security settings, such as the corruption threshold. Additional distinctions include whether security is based on computational assumptions or information-theoretic principles, as well as active versus passive adversaries. This work focuses on providing information-theoretic security against adversaries corrupting up to $\lfloor (n-1)/3 \rfloor$ parties, tolerating arbitrary deviations (active security).

We operate in the preprocessing model, where computation is split into an offline (preprocessing) phase and a subsequent, more efficient online phase. The preprocessing phase is independent of the parties' input data and generates correlated randomness for the online phase. We evaluate protocols and optimize with particular emphasis on *communication complexity* during the online phase, which is typically the primary bottleneck in practice.

MPC enables a broad range of privacy-preserving applications under the denominator computing on encrypted data (COED, [33]). Other applications include processing highly sensitive data in cases where processing in the clear is either prohibited by legal constraints or undesirable due to privacy concerns. In the case of COED, a client can secret-share their data and distribute designated shares to each party, requiring collusion of multiple parties to leak the original information. After receiving the shares, the parties jointly perform the computation and return the result to the client. The output can then be revealed to the client either in plaintext or as secret shares. This model enables stronger privacy guarantees than traditional cloud computation, while preserving data utility.

The cloud computing model is frequently used in conjunction with devices that cannot perform complex operations independently, such as Internet of Things (IoT) devices. These devices are seeing broad adoption popular in home, health, and manufacturing contexts. To ensure the smooth operation of a COED application with very lightweight sensors, the computational strain on the devices must be minimized. At the same time, there is a need for easy adaptability of deployed systems for data processing applications, including COED, which leads to designs leveraging transciphering (see, e.g., [1, 13, 32]). In transciphering-based architectures, the data is encrypted using a symmetric-key cipher rather than secret-shared. The ciphertext is sent to the MPC committee, and the symmetric key is threshold secret-shared among the parties. The NIST-standardized lightweight cryptography family Ascon is an excellent choice for transciphering, performing well on resource-constrained devices [38]. The Ascon family [18, 35] comprises authenticated symmetric-key encryption schemes designed for such environments. Ascon uses a modified sponge construction and an optimized substitution-permutation network (SPN) on a 320-bit state.

This paper investigates optimized approaches to compute Ascon in MPC in the information-theoretic, actively secure setting. While transciphering serves use cases beyond COED (e.g., enforcing consensus among key-holders for decryption), evaluating Ascon in MPC enables applications such as post-quantum signatures via MPC-in-the-Head (MPCitH)-based zero-knowledge (ZK) proofs [5, 24].

Not only do MPC and Ascon suit the aforementioned use cases, but the NIST (US) has also recently issued a call for multi-party threshold implementations of previously standardized primitives, including Ascon. According to the NIST, Multi-Party Threshold Cryptography (MPTC)[1] offers versatile applications by eliminating single points of failure for the security and availability of sensitive data, such as cryptographic keys. Importantly, the threshold implementation must retain full compatibility with the non-thresholded specification to ensure interoperability, as emphasized by NIST in its threshold call. Our study contributes to this project by investigating MPC implementations of Ascon.

We employ a Boolean circuit formulation of Ascon, as this is the most straightforward approach for secure MPC implementation.

1.1 Alternative Ciphers for Transciphering

Alternative ciphers for transciphering in our lightweight scenario are Arithmetic-Oriented ciphers (AO ciphers, [3, 4]) or other lightweight ciphers.

1.1.1 Arithmetic-Oriented Ciphers. AO ciphers are optimized for settings such as MPC by taking the mathematical structures underlying these settings into account. Thus, AO ciphers are, by design, more efficient in MPC. However, the bottleneck in our setting is the resource-constrained device. Furthermore, AO ciphers are typically unsuitable for lightweight devices due to the prohibitive overhead of large finite field arithmetic, as noted by Bhati [9]. This lack of alternatives creates a requirement for a lightweight cipher that performs well in MPC.

1.1.2 Other Ciphers. NIST's standardization of Ascon has established it as the leading cipher for lightweight devices. Furthermore, the confidence in the security of AO and other lightweight ciphers is substantially lower, because they have not received the same scrutiny that Ascon has, as noted by NIST [37, 38]. Moreover, the current lack of standardization efforts implies that it may take a long time before this situation improves. Additionally, Ascon's design leverages well-established components from NIST standards, such as SHA-2 and SHA-3, in its security-critical functions. Similar arguments appear in FAEST publications [5], particularly regarding ZK-based, quantum-resistant signatures. Following this, AES might also be considered for our use case. Nevertheless, performance and cost assessments in the NIST documentation indicate that AES is not feasible for our setting [38].

1.2 Contributions

The core contributions of this work are three-fold:

- **First MPC Optimization Study for Ascon:** We present the first investigation into optimizing Ascon for multi-party computation (MPC), inspired by NIST's threshold cryptography call and transciphering-based COED. Our approach includes a straightforward MP-SPDZ implementation designed for easy benchmarking in Sect. 3.
- **Optimal Technique Identification and Protocol Instantiation:** Through systematic evaluation, we identify Reverse-Multiplication Friendly Embedding (RMFE) as the most promising technique for Ascon in MPC. We determine suitable parameters and provide a concretely instantiated protocol, with modifications focused on the efficient execution of the linear diffusion layer within a single Ascon circuit using Network Routing in Sect. 4.
- **Theoretical Analysis:** We demonstrate a non-linear improvement in transmitted elements and bits for evaluating Ascon's permutation, dependent on the number of parties and RMFE packing size. This work provides an analysis of RMFE trade-offs and limitations in real-world workloads in Sect. 4.4 and Fig. 7.

1.3 Technical Overview

This section, provides a brief overview of Ascon and outlines the key aspects to optimize the MPC evaluation of Ascon, reserving a detailed description of each part for later sections.

Ascon. This paper focuses on Ascon-AEAD128. It is an authenticated encryption scheme with associated data. The scheme is constructed by evaluating the Ascon permutation in the sponge mode of operation. Since all sponge operations are linear concerning the internal state, the complexity of an MPC evaluation lies in the Ascon permutation.

The Ascon permutation operates on a 320-bit state $\mathcal{S}$ divided into five 64-bit words, denoted with S_i for $0 \leq i \leq 4$:

$$\mathcal{S} = S_0 \parallel S_1 \parallel S_2 \parallel S_3 \parallel S_4 .$$

Ascon's operations are geared towards single instruction, multiple data (SIMD) execution performed on the words S_i. The cipher's permutation p is applied in a round-based manner on the state. It consists of (1) addition of a round constant p_C, (2) a non-linear substitution layer (S-box, p_S), and (3) a linear diffusion layer p_L using rotations and XORs ($p = p_L \circ p_S \circ p_C$). The constant addition layer p_C adds a 64-bit constant described in the standard document as S_2: $S_2 = S_2 \oplus c_i$. The non-linear substitution layer, or S-Box p_S, applies the following function to state words, where $\cdot, \oplus$ denote AND and XOR evaluated bitwise over the 64-bit words:

$$\begin{aligned}
S_0' &= S_4 \cdot S_1 \oplus S_3 \oplus S_2 \cdot S_1 \oplus S_2 \oplus S_1 \cdot S_0 \oplus S_1 \oplus S_0 , \\
S_1' &= S_4 \oplus S_3 \cdot S_2 \oplus S_3 \cdot S_1 \oplus S_3 \oplus S_2 \cdot S_1 \oplus S_2 \oplus S_1 \oplus S_0 , \\
S_2' &= S_4 \cdot S_3 \oplus S_4 \oplus S_2 \oplus S_1 \oplus 1 , \\
S_3' &= S_4 \cdot S_0 \oplus S_4 \oplus S_3 \cdot S_0 \oplus S_3 \oplus S_2 \oplus S_1 \oplus S_0 , \\
S_4' &= S_4 \cdot S_1 \oplus S_4 \oplus S_3 \oplus S_1 \cdot S_0 \oplus S_1 .
\end{aligned}$$

While the five-bit S-Box function is effectively performed on a vertical one-bit wide slice taken from across the five 64-bit words, we are formulating the function as taking complete 64-bit words. The linear layer p_L consists of the following functions being applied

[1] https://csrc.nist.gov/projects/threshold-cryptography

to state words:

$$\begin{aligned}
\Sigma_0(S_0) &= S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28)\,,\\
\Sigma_1(S_1) &= S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39)\,,\\
\Sigma_2(S_2) &= S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6)\,,\\
\Sigma_3(S_3) &= S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17)\,,\\
\Sigma_4(S_4) &= S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41)\,,
\end{aligned} \tag{1}$$

where $S_i \ggg j$ denotes rotating bits in S_i by j places to the right. Note that the linear layer operates on bits inside individual words.

Applying Packing for SIMD Operations. The main idea of our work is to utilize packing techniques in secret sharing-based MPC protocols to minimize the communication complexity of the MPC computation for the non-linear substitution layer. Ideally, we pack each 64-bit word S_i into a single share $[S_i]$ and then compute the S-box as

$$\begin{aligned}
\left[S_0'\right] &= [S_4]\cdot[S_1] + [S_3] + [S_2]\cdot[S_1] + [S_2]\\
&\quad + [S_1]\cdot[S_0] + [S_1] + [S_0]\\
\left[S_1'\right] &= \dots
\end{aligned}$$

Note that this can be done with only five multiplications of packed sharings instead of $5 \cdot 64$ bit-bit multiplications. We discuss relevant SIMD-based MPC protocols in Sect. 4.1.2 and settle on reverse multiplication-friendly embeddings (RMFE). RMFEs embed a vector of elements from a small field into a single element in a larger field. RMFEs are a substitute for an isomorphism between vectors of elements $\mathbb{F}_2^k$ and scalars $\mathbb{F}_{2^k}$, as such an isomorphism cannot exist. While the embeddings do not form an isomorphism, they are easy to work with in an MPC protocol. A $(k, m)_q$-RMFE with $k, n \in \mathbb{N}_{\geq 1}$ and q a prime power consists of a pair (ϕ, ψ) where $\phi : \mathbb{F}_2^k \to \mathbb{F}_{2^m}$ and $\psi : \mathbb{F}_{2^m} \to \mathbb{F}_2^k$ are $\mathbb{F}_q$ linear maps with $\mathbf{x} * \mathbf{y} = \psi\big(\phi(\mathbf{x}) \cdot \phi(\mathbf{y})\big)\ \forall\, \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^k$.

Applying this idea to the remaining Ascon circuit introduces two challenges. First, since $m > k$, we need to find concrete parameters for k, m for which the MPC multiplication of two batched shares has an improved communication complexity compared to bit-bit multiplication. We describe such parameters in Sect. 4.2. While we don't find a useful setting where $k \geq 64$, i.e., that packs all bits of one word into one share, our parameters pack three or four bits per share, which still reduces the communication cost.

Second, the bit-rotation in the linear diffusion layer $[S_i] \ggg j$ is no longer a linear operation. We adapt network-routing techniques from Goyal et al. [23]. In short, our solution makes use of shares $[\vec{r}]\,, [L_1(\vec{r})]\,, [L_2(\vec{r})]$ of a random vector of values $\vec{r}$, where L_1, L_2 denote the desired linear operations (in our case, the bit-rotation) that parties obtain from the preprocessing phase. Then given a packed share of the i-th word $[S_i]$, one party reconstructs $S_i + r$, locally computes the linear operation and broadcasts the rotated and blinded value to all parties. The parties locally obtain the desired rotation as $L(S_i + r) - [L_1(r_i)] - [L_2(r_i)]$.

1.4 Related Work

To the best of our knowledge, only Mandal and Gong [28] study the performance of lightweight ciphers, including Ascon, in secure computation. However, they use a garbled circuit approach and an FHE protocol without optimizations. Following this, we discuss approaches to optimizing AES and other ciphers from the broader research field of symmetric-key primitives in an MPC context.

1.4.1 AES Optimizations. Delpech de Saint Guilhem et al. [16] optimize AES evaluation in MPC-in-the-Head to construct a signature scheme. Chida et al. [12] search for small features in AES that can be optimized for MPC; they also optimize the general framework of MPC protocol execution and ensure that essential functions of the MPC protocol fit well with the overall AES structure. This includes generating values on the fly and choosing a specific multiplication protocol, as well as combining specific steps in the AES round function. In a follow-up work in the same setting, MAESTRO [29] improves finite field inversion used in the AES S-box by utilizing oblivious lookup table protocols. Although the setting of Chida et al.'s work differs from ours, we also choose a specific multiplication protocol for the steps in the Ascon round function.

Damgård and Zakarias [15] optimize the evaluation of AES with MPC with a construction called Mini-Mac, an approach that will be evaluated in Sect. 4.1.2. This line of work as well as the original packing approach by Franklin and Yung [22] are evaluated in Sect. 4.1.1.

1.4.2 Lightweight Cipher Optimization. The ciphers mentioned here are lightweight ciphers or have a similar structure to Ascon. Stoffelen [36] optimizes the S-Boxes of lightweight ciphers, including Ascon, using SAT solvers. However, they do not regard the permutation as a whole, and the number of optimization-critical non-linear operations performed in the S-Box has not decreased. An independent follow-up work by Zhang and Huang [39] revisits the question, but without relevant improvements for Ascon.

Abidin et al. [2] provide optimized arithmetic expressions for the lightweight ciphers Skinny and Photon. Such an efficient, dedicated arithmetic expression is, as the authors of the paper state, hard to come by for Ascon, due to the different ways in which Ascon's linear and non-linear layers treat Ascon's internal state.

2 Background on Ascon and RMFEs

In the following, we present background details on the Ascon lightweight block cipher family (Sect. 2.1) and RMFEs (Sect. 2.2) as well as an instantiated version of an online protocol based on previous work (Sect. 2.3).

2.1 Ascon

Ascon is a family of lightweight ciphers and hashing schemes proposed by Dobraunig et al. [18], chosen by the NIST for standardization in the lightweight cryptography competition [38]. It was chosen because it has the most extensive security analyses, very good performance in software and hardware, and flexibility in trading off costs and performance through various implementations. These properties, as well as their standardization, make Ascon the ideal contender for future uses in lightweight devices, such as smart cards or sensors.

The Ascon proposal submitted to the lightweight competition includes hash and extendable output functions (XOFs) as well as three parameter sets for authenticated encryption. The current NIST specifications (see [35]) focus on the parameter set Ascon-AEAD128 for authenticated encryption with associated data (AEAD), referred

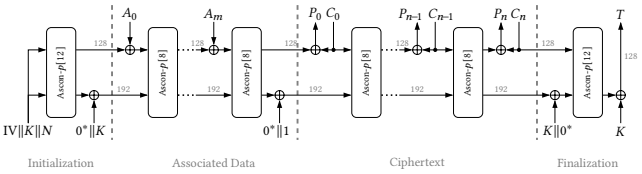


Figure 1: Ascon decryption mode. Figure from [18].

to in submission documents as Ascon-128a, which we also focus on in this work.

Ascon processes variable-length associated data A, plaintext P (or ciphertext C), each divided into 128-bit blocks, together with a 128-bit key K and nonce N. The output consists of either the corresponding plaintext or ciphertext along with a 128-bit authentication tag T. The tag T is computed during encryption and verified during decryption. Except for the finalization phase, the encryption and decryption modes are constructed in an identical fashion.

Ascon's structure is based on a sponge construction, as shown in Fig. 1 for the decryption mode. At its core, Ascon has a 320-bit state divided into five 64-bit words, with S_i for $0 \leq i \leq 4$ s.t. $\mathcal{S} = S_0 \parallel S_1 \parallel S_2 \parallel S_3 \parallel S_4$.

The cipher's permutation p is applied to the state in a round-based manner. During block processing, eight permutation rounds are executed per block, while during initialization and finalization, 12 rounds are executed. Further, the permutation consists of a constant layer p_C, a substitution layer p_S and a linear layer p_L executed consecutively ($p = p_L \circ p_S \circ p_C$).

2.1.1 Constant Layer p_C. In the constant layer p_C, a constant value dependent on the current permutation round is XOR'd with a part of the word S_2.

2.1.2 Substitution Layer p_S. The substitution layer p_S modifies the state S by applying the substitution box to 64 parallel inputs, as follows:

$$(S_0, S_1, \ldots, S_4) = \text{S-Box}(S_0, S_1, \ldots, S_4)\,,$$

for $0 \leq j < 64$.

The S-Box as re-formulated here takes 5 64-bit inputs $S = (S_0, S_1, \ldots, S_4)$ and generates 5 64-bit outputs using the circuit shown in Fig. 2.

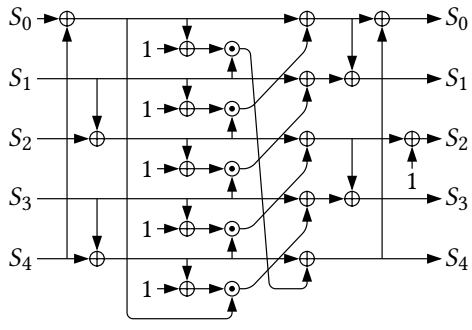


Figure 2: Ascon S-Box. Figure from [18].

2.1.3 Linear Layer p_L. The linear diffusion layer p_L applies the linear functions Σ_i to their corresponding state words as shown in Eq. (1), updating them as $S_i \leftarrow \Sigma_i(S_i)$ for $0 \leq i \leq 4$. Each function Σ_i is defined by two circular bit rotations on the state word.

2.2 Reverse Multiplication Friendly Embeddings

In their 2018 paper [11], Cascudo et al. introduce Reverse Multiplication Friendly Embeddings (RMFE) as a novel method for packing secret-shared values into single shares. Their construction ensures perfect security against active adversaries in the $n = 3t + 1$ setting, where n denotes the number of parties and t the number of tolerated corruptions. As in prior packing approaches [22], RMFEs enable simultaneous evaluation of multiple instances of the same circuit. The novelty lies in trading an increased field size for greater parallelism, contrasting with earlier packed secret sharing techniques that traded field size for adversarial threshold. The authors focus on binary circuit evaluation because extension fields result in a huge communication overhead for this variant.

Specifically, this overhead arises from secret sharing protocols requiring unique shares per party. For $\mathbb{F}_2$: extension fields provide these unique shares by mapping computational values to shares. However, transmitting a share of size five bits instead of the single bit needed for computation creates substantial overhead. This overhead increases as the number of participants grows: since a larger number space is required to support a greater quantity of unique shares. Consequently, the size of each share also increases. To amortize this overhead, the goal becomes leveraging the $\mathbb{F}_2$ to $\mathbb{F}_{2^k}$ transition by performing k simultaneous binary computations. The ideal mapping would be an isomorphism between $\mathbb{F}_2^k$ and $\mathbb{F}_{2^k}$, but no such isomorphism exists for $k \geq 2$ (because $\mathbb{F}_2^k$ has zero divisors [2]).

As a substitute for the ideal mapping, Cascudo et al. define RMFEs and a protocol that performs the necessary mappings and operations. The operations for the mapped values are computed using a secondary arithmetic-circuit protocol, π'. This protocol is regarded as a black box in the 2018 paper, but we will instantiate it concretely with the protocol introduced by Beerliová-Trubíniová and Hirt in their 2008 paper [8]. A compiler can then provide the necessary mappings and modifications to perform packings and transform the relevant operations from a simple Boolean circuit C to the packed arithmetic circuit C'. In the RMFE-based protocol, a parallelized circuit is thus compiled to an arithmetic representation rather than a Boolean one. This approach avoids a dedicated arithmetic formulation. In the following, RMFEs will be defined, and two practical constructions will be given.

2.2.1 Defining RMFEs. As stated earlier, RMFEs are a substitution for the ideal mapping between $\mathbb{F}_2^k$ and $\mathbb{F}_{2^k}$. The definition for RMFEs is as follows:

Definition 2.1 (Reverse Multiplication-Friendly Embedding [11]). Let $\mathbb{F}_q$ be a finite field of q elements, where q is a power of a prime, and let $k, m \geq 1$ be integers. We call a pair (ϕ, ψ) a $(k, m)_q$-RMFE if

[2] For example, in $\mathbb{F}_2^2$, the vectors $(1, 0)$ and $(0, 1)$ are both nonzero, but their componentwise product is $(0, 0)$, illustrating the presence of zero divisors.

$\phi : \mathbb{F}_q^k \to \mathbb{F}_{q^m}$ and $\psi : \mathbb{F}_{q^m} \to \mathbb{F}_q^k$ are two $\mathbb{F}_q$-linear maps satisfying

$$\mathbf{x} * \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y})), \forall\, \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^k\,.$$

For this work, we focus on $q = 2$.

2.2.2 Practical RMFE Constructions. Cascudo et al. present direct constructions of small RMFEs and specify concatenation techniques to build larger RMFEs. For packing binary vectors in $\mathbb{F}_2^k$ with extension field degrees up to 325, they establish the following explicit constructions:

- A $(2r, 6r-3)_2$-RMFE for $r \le 9$, formed by concatenating a $(2,3)_2$-RMFE and an $(r, 2r-1)_8$-RMFE.
- A $(3r, 10r-5)_2$-RMFE for $r \le 33$, constructed via concatenation of a $(3,5)_2$-RMFE and an $(r, 2r-1)_{32}$-RMFE.

This approach scales RMFEs while maintaining algebraic structure constraints.

$(3,5)_2$-RMFE Construction. The $(3,5)_2$-RMFE is a building block that, unlike larger RMFEs, does not require chaining of smaller embeddings. We define the pair (ϕ, ψ) as follows:

$$\phi = \mathcal{E}_2 \circ \mathcal{E}_1^{-1} : \mathbb{F}_2^3 \to \mathbb{F}_{2^5}\,,$$
$$\psi = \mathcal{E}_{1'} \circ \mathcal{E}_2^{-1} : \mathbb{F}_{2^5} \to \mathbb{F}_2^3\,.$$

Core Mappings:

(1) $\mathcal{E}_1 : \mathbb{F}_2[X]_{\le 2} \to \mathbb{F}_2^3$. For $f(X) = aX^2 + bX + c$:
$$\mathcal{E}_1(f) = \big(f(0),\, f(1),\, a\big) = (c,\, a+b+c,\, a)\,.$$
(2) $\mathcal{E}_2 : \mathbb{F}_2[X]_{\le 4} \to \mathbb{F}_{2^5}$. Let α be a root of the irreducible polynomial $P(X) = X^5 + X^2 + 1$ over $\mathbb{F}_2$: $\mathcal{E}_2(g) = g(\alpha) \in \mathbb{F}_{32}$.
(3) $\mathcal{E}_{1'} : \mathbb{F}_2[X]_{\le 4} \to \mathbb{F}_2^3$. For $g(X) = \sum_{i=0}^4 g_i X^i$: $\mathcal{E}_{1'}(g) = \big(g(0),\, g(1),\, g_2\big)$.

Inverse Mappings:

(1) $\mathcal{E}_1^{-1} : \mathbb{F}_2^3 \to \mathbb{F}_2[X]_{\le 2}$. For $\mathbf{v} = (v_1, v_2, v_3)$: $f(X) = v_3X^2 + (v_1 + v_2 + v_3)X + v_1$.
(2) $\mathcal{E}_2^{-1} : \mathbb{F}_{2^5} \to \mathbb{F}_2[X]_{\le 4}$. For $\beta \in \mathbb{F}_{32}$, compute coefficients c_i where $\beta = \sum_{i=0}^4 c_i \alpha^i$: $g(X) = \sum_{i=0}^4 c_i X^i$.

This explicit construction provides a practical instantiation of the $(3,5)_2$-RMFE, with all mappings and their inverses given in closed form.

Composition:

- $\phi(\mathbf{v})$: Map $\mathbf{v}$ to $f \in \mathbb{F}_2[X]_{\le 2}$, then apply $\mathcal{E}_2$.
- $\psi(\beta)$: Map β to $g \in \mathbb{F}_2[X]_{\le 4}$ via $\mathcal{E}_2^{-1}$, then apply $\mathcal{E}_{1'}$.

Example: For $\mathbf{v} = (1,0,1)$: $\mathcal{E}_1^{-1}(\mathbf{v}) = X^2 + 1 \xrightarrow{\mathcal{E}_2} \alpha^2 + 1$. For $\beta = \alpha^2 + 1$: $\mathcal{E}_2^{-1}(\beta) = X^2 + 1 \xrightarrow{\mathcal{E}_{1'}} \big(1,\, 1^2+1=0,\, 1\big) = (1,0,1)$.

$(4,9)_2$-RMFE Construction. A $(4,9)_2$-RMFE can be constructed by instantiating the general $(2r, 6r-3)_2$-RMFE construction with $r = 2$. The construction proceeds in two stages:

(1) **First stage:** Partition the input vector $\mathbf{x} = (x_1, x_2, x_3, x_4) \in \mathbb{F}_2^4$ into two pairs: (x_1, x_2) and (x_3, x_4). Each pair is embedded into $\mathbb{F}_8$ via a $(2,3)_2$-RMFE. The inverse mapping ψ_1 recovers (a, b) from an element of $\mathbb{F}_8$.
(2) **Second stage:** The resulting vector $(y_1, y_2) \in \mathbb{F}_8^2$ is encoded into $\mathbb{F}_{2^9}$ using a $(2,3)_8$-RMFE. The inverse mapping ψ_2 expresses an element of $\mathbb{F}_{2^9}$.

The overall embedding map is thus

$$\phi = \phi_2 \circ (\phi_1 \times \phi_1) : \mathbb{F}_2^4 \to \mathbb{F}_{2^9}\,,$$

where $\phi_1 \times \phi_1$ applies ϕ_1 to each pair. The decoding map is

$$\psi = (\psi_1 \times \psi_1) \circ \psi_2 : \mathbb{F}_{2^9} \to \mathbb{F}_2^4\,.$$

For all $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^4$, the component-wise product satisfies $\mathbf{x} * \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$, as required for an RMFE. Building upon the RMFEs introduced previously, we are going to give an instantiated online protocol in the following.

2.3 Instantiation of the Online Protocol

In this subsection, we present the instantiation of our RMFE-based online protocol for Ascon execution. We present a high-level view of the protocol construction, specify our choice of the underlying arithmetic circuit protocol, and describe the core modifications made to support efficient network routing and parallelization. Parameter choices will be discussed in Sect. 4.2: Additional sub-protocols regarding value reconstruction in the online protocol are provided in Appendix A.1.

The protocol follows Cascudo et al. [11], which uses an arithmetic circuit protocol π' as a black box. We instantiate this black box protocol with the protocol of Beerliová-Trubíniová and Hirt [8]. We then integrate an RMFE packing mechanism and incorporate network routing via cyclic right rotations to enable parallelization of non-SIMD circuits. Together with the SIMD-adjusted Ascon evaluation specified in Sect. 4, these elements form our concrete RMFE-based online protocol for evaluating Ascon. While preprocessing is not the focus of this work, it should be noted that a functionality (called $\mathcal{F}_{\text{RandElSub}}$ by Cascudo et al.) is used to generate random elements from the image of ϕ for ensuring correctness of inputs as well as re-encoding values using ϕ after multiplications.

2.3.1 Secret Sharing. The choice of the underlying protocol specified in Sect. 2.3 determines the underlying secret sharing scheme and provides the foundation for our implementation. The sharing of a general scalar element x in our protocol will be denoted as $[x]$. The sharing of a vector $\vec{x} = (x_1, \ldots, x_k)$ mapped via RMFEs will be denoted as $[\phi(\vec{x})]$. In contrast to the vectors, the general elements are used specifically in parts where a concrete vector mapping is not explicitly used in the protocol.

2.3.2 Main Online Protocol. Figure 3 outlines the main phases of the RMFE-based online protocol. The following paragraphs summarize each phase to convey the protocol's core ideas succinctly.

Input Phase. Each party P_i with input $\vec{x}_i = (x_i^{(1)}, \ldots, x_i^{(k)}) \in \mathbb{F}_2^k$ first maps its input using ϕ, obtaining $\phi(\vec{x}_i)$. To share this value, the protocol privately reconstructs a preprocessed sharing $[r]$ to P_i, which then broadcasts the blinded difference $d = \phi(\vec{x}_i) - r$. All parties locally compute their shares as $[\phi(\vec{x}_i)]_j = d + r_j$. To ensure mapping correctness, a fresh random sharing $[r']$ of an element $r' \in \operatorname{Im}\phi$ is added locally, and the sum $[\phi(\vec{x}_i) + r']$ is publicly reconstructed. The input is accepted only if the revealed value lies in $\operatorname{Im}\phi$; otherwise, it is rejected. The correctness guarantee comes from the fact that $\operatorname{Im}\phi$ is a $\mathbb{F}_2$-vector subspace, so that if $r', \vec{x}_i + r \in \operatorname{Im}\phi$, then $\vec{x} \in \operatorname{Im}\phi$. This prevents adversaries from introducing invalid shares outside the image of ϕ.

Addition. Addition is performed locally: each party adds its shares $[a_i]$ and $[b_i]$ to obtain $[a_i + b_i]$. To add constants, treat them as "0-sharings" and proceed analogously.

Multiplication. Multiplication is performed with the aid of batches of Beaver triples $[a_k], [b_k], [c_k]$, where $c_k = a_k b_k$, as introduced by Beaver [6]. These preprocessed triples allow multiplication to be reduced to linear operations and a single public reconstruction step, ensuring security and efficiency. For each pair of inputs, parties compute the differences $[d_k] = [x_k] - [a_k]$ and $[e_k] = [y_k] - [b_k]$, then publicly reconstruct (d_k, e_k). The product share is computed locally as $[z'_k] = d_k e_k + d_k[b_k] + e_k[a_k] + [c_k]$. To ensure the result is a valid RMFE-mapped sharing, a fresh correlated sharing $([r], [s])$ with $s = \phi(\psi(r))$ is used: parties locally add $[z'_k] + [r]$, reveal the sum, and output $[\phi(\psi(z'_k))]$ = revealed value $- [s]$.

Output. To reconstruct the output, the designated party receives $[\phi(\vec{z})]$ via private reconstruction. It then applies the inverse RMFE mapping to recover the result: $\mathbf{y} = \phi^{-1}(\vec{z})$.

2.3.3 Network Routing. Previous packing-based works that consider single-circuit execution with SIMD elements usually consider general circuits, i.e., circuits that are not purely SIMD and are built on top of amortized-cost packing approaches. When using packing techniques previously employed for SIMD-parallelized execution of multiple instances, the problem of re-ordering gate outputs arises. This problem comes from the possibility that values packed into one share might have to be re-ordered to match the correct gates at the next circuit layer. To fix this, a technique called "network-routing" was developed.

While the Ascon decryption does not require general reordering between layers, bit shifts still necessitate network routing. The protocol has similarities to parts of the network routing subprotocols by Goyal et al. and Escudero et al., but is more specific, focusing on permuting a fixed number of shared values into new sharings corresponding to that permutation [20, 23].

Protocol `LinearTrans`. At a high level, the `LinearTrans` protocol allows parties to apply a linear transformation L to vectors of packed, secret-shared values $[\phi(\vec{x})]$ to obtain $[\phi(L(\vec{x}))]$. To facilitate this, the protocol requires preprocessing of a pair of random values $[\phi(r)]$ and their corresponding transformed random values $[\phi(L(r))]$. The protocol operates by first blinding the input sharings, then publicly reconstructing the masked values, and applying the linear transform on the revealed value in the clear. Afterwards, the value "blind" is removed using the second component of the preprocessing values. This methodology ensures both privacy and correctness, while enabling flexible data movement in the packed secret sharing setting. For a formal description, see Fig. 4.

3 Straightforward Implementation of Ascon in MPC

This section, presents a straightforward implementation of Ascon for easy benchmarking. For this, we first outline theoretical considerations regarding how the Ascon decryption function must be adapted for an MPC setting. Afterward, we describe a concrete implementation using the MP-SPDZ framework [25].

RMFE-based online protocol

Input

Input: Party P_i inputs $\vec{x}_i$.
Output: $[\phi(\vec{x}_i)]$, with mapping in $\mathrm{Im}\,\phi$.

(1) Reconstruct the sharing $[r]$ from preprocessing towards P_i using `ReconsPriv`. Robustness is ensured through $t < n - 2t$.
(2) P_i applies ϕ, blinds and broadcasts $d = \phi(\vec{x}_i) - r$.
(3) Every $P_j; j \neq i$ locally computes $[\phi(\vec{x}_i)]_j = d + \lfloor r_j \rfloor$ to acquire their share.
(4) Select the next fresh sharing $[r']$ generated in preprocessing.
(5) Perform a local computation of $[\phi(\vec{x}_i) + r'] = [\phi(\vec{x}_i)] + [r']$.
(6) Reveal $[\phi(\vec{x}_i) + r']$ to all parties using `ReconsPubl`. If the result $\in \mathrm{Im}\,\phi$, accept mapped sharing $[\phi(\vec{x}_i)]$. Otherwise, reject.

Addition

Addition is performed by each player P_i locally adding their shares $[a_i]$ and $[b_i]$ to get $[a_i + b_i]$. In the case of constants, one can simply treat them as "0-sharings" and perform the addition as usual as in the blackbox protocol as per Beerliová-Trubíniová and Hirt [8].

Multiplication

Input:$([\phi(\vec{x}_1)], [\phi(\vec{y}_1)]), \ldots, ([\phi(\vec{x}_{T/2})], [\phi(\vec{y}_{T/2})])$ with $T = n - 2t$;
Output: $[\phi(\vec{z}_1)], \ldots, [\phi(\vec{z}_{T/2})]$

(1) Take fresh $([a_1], [b_1], [c_1]), \ldots, ([a_{T/2}], [b_{T/2}], [c_{T/2}])$ with $c_p = a_p b_p$ for $p = 1, \ldots, T/2$ from preprocessing.
(2) Each player, for $p = 1, \ldots, T/2$, calculates $[d_p] = [\phi(\vec{x}_p)] - [a_p]$ and $[e_p] = [\phi(\vec{y}_p)] - [b_p]$.
(3) Reveal the T sharings $(d_1, e_1), \ldots, (d_{T/2}, e_{T/2})$ using ReconsPubl.
(4) Each player, for $p = 1, \ldots, T/2$, determines the product sharings $[z'_p] = de + d[b] + e[a] + [c]$.
(5) Take fresh correlated sharing $([r], [s])$ with $s = \phi(\psi(r))$.
(6) Each party computes $[z'_p + r] = [z'_p] + [r]$ locally.
(7) Reveal $y = [z'_p + r]$ using `ReconsPubl`.
(8) Output $[\phi(\vec{z}_p)] = \phi(\psi(z'_p)) - [s]$.

Output

Input: $[\phi(\vec{z})]$.
Output: $\vec{z}$.

(1) Reconstruct $[\phi(\vec{z})]$ to towards the designated party to receive $\phi(\vec{z})$ using `ReconsPriv`.
(2) Output $\phi^{-1}(\phi(\vec{z}))$.

Figure 3: Online phase of the RMFE based Protocol merged/adapted from [8, 11].

LinearTrans

Input: $[\phi(\vec{x}_1)], \ldots, [\phi(\vec{x}_M)]$ input sharings, $L_1, \ldots, L_M$ linear mappings, $M = n - 2t$.
Output: $\left[\phi\left(L_p\left(\vec{x}_p\right)\right)\right]$ for all $1 \leq p \leq M$.

(1) All parties take fresh correlated random sharings from preprocessing: $\left(\left[\phi\left(\vec{r}_1\right)\right], \left[\phi\left(L_1\left(\vec{r}_1\right)\right)\right]\right), \ldots, \left(\left[\phi\left(\vec{r}_M\right)\right], \left[\phi\left(L_M\left(\vec{r}_M\right)\right)\right]\right)$.
(2) The parties call publicly reveal the sharings $\left[\phi\left(\vec{x}_p\right)\right] + \left[\phi\left(\vec{r}_p\right)\right]$ for $p = 1, \ldots, M$ using `ReconsPubl`, obtaining $\phi\left(\vec{x}_p\right) + \phi\left(\vec{r}_p\right)$.
(3) Each party applies $\psi(\cdot)$ locally to each revealed value to recover $\vec{x}_p + \vec{r}_p$.
(4) Compute locally $\phi\left(L_p\left(\vec{x}_p + \vec{r}_p\right)\right)$, and output the sharing $\left[\phi\left(L_p\left(\vec{x}_p + \vec{r}_p\right)\right)\right] - \left[\phi\left(L_p\left(\vec{r}_p\right)\right)\right]$ for each $p = 1, \ldots, M$.

Figure 4: Protocol `LinearTrans` adapted from [20].

Theoretical Considerations. For the baseline, we implement Ascon using the natural Boolean function design. The class of MPC protocols we aim at incurs communication costs for AND gates between two secret-shared values, while XOR gates can be computed without communication overhead. Additionally, bit reordering operations can be performed efficiently and locally if we secret-share individual bits. Operations involving public inputs require no communication.

To cover a broad range of use cases, we treat the plaintext as a private output and the key K as secret. In contrast, we consider the nonce, tag, ciphertext, and associated data (N, T, C, A) as public.

The individual bits forming private words are secret-shared in this implementation. Since the bits of 64-bit words are secret-shared individually, operations such as AND and XOR are executed bitwise, with 64 gates operating in parallel. This approach applies similarly to other operations and data objects. This representation leads to the implementation of the fundamental operations described below. We keep public data as normal bit-words, consistent with the non-MPC Ascon standard documents.

We can perform other operations listed in the Ascon reference without communication, as they are either manipulation of registers (e.g., reshuffling) or operations on public data.

Permutation. The permutation is decomposed into components $p = p_L \circ p_S \circ p_C$. Among these, only the substitution layer p_s involves five AND gates and thus requires communication.

Decryption Process. When considering the decryption/encryption process, it is clear that only the permutation uses the AND operation and hence incurs communication overhead. Thus, the only stages and operations in the decryption process requiring communication are, for our chosen parameter set $a = 12, b = 8$:

(1) Initialization: $S \leftarrow p^a(S) \oplus (0^{320-k}||K)$, applies the permutation a times;
(2) Processing Associated Data: $S \leftarrow p^b((S_r \oplus A_i)||S_c)$, applies the permutation b times for each r-bit block A_i of associated data;
(3) Processing Ciphertext Data: $S \leftarrow p^b(S)$, applies the permutation b times for all but the last r-bit blocks C_i of ciphertext;
(4) Finalization: $S \leftarrow p^a(S \oplus (0^r||K||0^{320-r-k}))$, applies the permutation a times.

The main difference compared to a software implementation of Ascon's encryption and decryption process lies in the data representation and the modified bitstring operations, which involve reordering and discarding bit-shares. Additionally, a reveal of secret-shared data is necessary to compare the computed tag against the received tag and to reveal the plaintext if desired.

3.1 Implementation

This section presents an implementation of the baseline approach using MP-SPDZ [25]. We first describe its data types and relevant details, followed by implementation specifics.

3.1.1 MP-SPDZ. MP-SPDZ is a widely recognized framework for developing and benchmarking MPC applications across multiple protocols.

In the following, we list the MP-SPDZ types used.

- `cbits`: Register of clear bits supporting basic bitwise operations. A single clear bit is represented by a register of length one.
- `sbit`: Single secret bit.
- `sbits`: Register of secret bits supporting basic bitwise operations.

These types provide a `get_type` method to obtain a register type of a specified length [25].

3.1.2 Implementation Details. The implementation of the baseline approach for Ascon in MP-SPDZ closely follows the Ascon reference implementation. The main differences are in the data types used and the MP-SPDZ-specific control structures, which are detailed below. We give preliminary benchmark results in Table 5 in Appendix A.4 for three Boolean circuit MPC protocols supported by MP-SPDZ.

The state S is represented as an array of five `sbits`, with other private data objects represented similarly. In this figure, the data types of the corresponding sizes are represented by, e.g., "`sb64`" denoting a `sbits` type of length 64. The return values denote the plaintext (length N), the state (length five), and, in the Finalize function, the computed tag (length 128).

If a data type, such as the state S, is represented as a list of, e.g., "`sb64`" with length five, it is denoted as "`[sb64*5]`", with other data types displayed correspondingly. Furthermore, XOR and AND gates are directly applied to the corresponding `cbits` and `sbits` of appropriate length using the integrated operations these types provide—this makes the 64-bit word-wise operations straightforward.

Another difference from the Ascon reference implementation is the ROTR method, that we adapted to use the `bit_decompose` method provided by the `cbits` and `sbits` classes, performing array slicing on the results and recomposing them into the desired data types.

We largely retain the control flow and method sequence from the reference, replacing them with MP-SPDZ control flow macros where necessary. Further, we provide a listing with the corresponding data types and permutation in Appendix A.5.

4 Optimized Ascon in MPC

In this section, we first discuss approaches to optimize Ascon's representation in MPC in Sect. 4.1. Then, we choose parameters for our Ascon-RMFE protocol in Sect. 4.2 and describe the protocol itself in Sect. 4.3. To conclude, we evaluate the protocol performance in Sect. 4.4.

4.1 Evaluation of Optimization Approaches

This section explores the potential optimizations for Ascon, with a special regard to works that did not consider Ascon specifically. Specifically, the focus lies on the Ascon permutation and the S-Box, as they are the most costly parts in the straightforward implementation. As mentioned earlier, optimization of computational overhead will be disregarded here, as it is not the weak point of MPC ([21], p.7). Preprocessing costs will also not be considered here, and the material from preprocessing will be taken as given.

Ascon has a very simple S-Box with low multiplicative depth. Furthermore, it does not need to rely on lookup tables for an efficient S-Box implementation. These facts close off some optimization routes previously chosen for AES and other ciphers, as discussed in the following.

4.1.1 Optimization Methods. We first discuss general approaches to S-box optimization and afterwards describe MPC protocols for performing SIMD operations.

Optimized S-Box Expressions in Binary Circuits. The 2016 and 2023 works by Stoffelen, as well as Zhang and Huang [36, 39] search for optimized S-Box expressions for lightweight ciphers using satisfiability checking approaches. Multiple expressions are given for the S-Boxes, including Ascon, optimizing for different metrics. However, the expressions do not allow for significant cost reduction in terms of multiplicative depth in the MPC setting due to Ascon's S-Box relying on linear operations and only a single layer of non-linear operations. More complex formulations, batching permutation rounds together, could lead to a reduction of multiplicative depth, but they will result in very unwieldy formulations due to the cipher rounds being designed to make output bits dependent on as many inputs as possible. Hence, this direction will not be considered or explored further in this work.

Table Lookup-based Approaches. Oblivious table lookup approaches, such as [10, 17, 26, 29], as used in AES optimization, will likely not yield good results for Ascon, since Ascon has a small 5-bit S-box compared to an 8-bit S-box for AES. Table-lookups would also not utilize the inherent parallelism of the 64-bit word operation execution recommended in the Ascon specification. Furthermore, table lookups in MPC are associated with additional costs that have to be carefully balanced against plain multiplication/AND operation. Following this, lookup tables will not be investigated further in this work.

Arithmetic Formulations. When applying the polynomial decomposition approach by Abidin et al. [2] to Ascon's 5-bit S-box, its evaluation cost is three multiplications in $\mathbb{F}_{2^5}$ per S-box. Since now, the substitution layer is composed of 64 such S-boxes, the total cost per substitution layer is 192 multiplications in $\mathbb{F}_{2^5}$. The encoding of the permutation state into shares for this approach also suffers from the problem that the bit-rotations of the linear layer are also no longer linear operations. Here, it is not clear how the rotation can be computed without broadcasting all state bits in individual shares which would undermine the saved communication cost of the substitution layer. Thus, SIMD-based approaches seem to be more attractive.

SIMD-based Approaches. Hardware implementations of Ascon are recommended [18] to operate as a single input, multiple data (SIMD) over the words of the SPN state. To use this structure in MPC-based Ascon evaluation, SIMD-optimized works will be laid out hereafter.

4.1.2 SIMD MPC Research Landscape. The SIMD and thus packing-based approaches listed below were built to parallelize the execution of the same circuit on multiple inputs. These approaches are employed to amortize the costs of processing multiple instances of the same circuit for, e.g., multiple users requesting the same function evaluation from an MPC committee. All the approaches employ some form of packing, where k values are packed into a share instead of one value, with the function being computed on packed values. After the computation, the result is unpacked and passed on, e.g., to the end-user.

Our focus is on how these approaches can be adapted to a single-circuit execution with SIMD elements. As Ascon is built to be executed in a SIMD fashion in hardware, it might be possible to pack its values in a way that is compatible with this design.

As detailed in Sect. 2.3.3, packing based evaluations of general single circuits with need to be able to handle re-ordering of gate outputs to match the input order of gates of the next layer. These are equivalent to those circuit parts that would, in hardware, not be purely SIMD, but rather requiring reconstruction or loading of registers.

Ascon's operation is nearly completely SIMD. However, the permutation layer requires re-ordering of the 64-bit words of the state. In the conventional approach, one can take bit-shares and re-index them as required. Thus, this re-ordering problem has to be solved in the packed SIMD setting. The works listed below take similar approaches to addressing the problem.

Original Packing Approach. Following the work of Franklin and Young from 1992, [22], a line of work in MPC research uses a single share to effectively represent multiple packed shares. These packings rely on more points for secrets to be stored on the polynomial than in traditional Shamir secret sharing [34]. Initially intended for pure SIMD circuits that were formed by simply executing the same circuit in parallel, it has since been extended to single-circuits with SIMD elements [7, 19, 23]. However, this packing approach requires trading off packing density against the corruption threshold due to the construction of the polynomial for secret sharing. This is a significant disadvantage for a general-purpose implementation and will thus not be considered further.

MiniMAC. The 2013 work by Damgård and Zakarias allows for packing, but in the computational and active security setting. In this setting, specialized tools are required in contrast to the approach listed before, as MAC (Message Authentication Codes) checks have to be performed correctly in addition to plain MPC evaluation. This MAC-checking requires, according to the authors, additional

effort when applied in the packing approach. The work proposes to resolve this by encoding the packed values in error-correcting codes, for example, Reed-Solomon codes. Our initial evaluation has shown, however, that compared to other non-packing approaches in the same security setting, the performance does not scale well. While the packing allows for parallelism, it necessarily introduces great overhead due to the code-based encoding. This overhead and the use of amortized checks by other protocols such as the TinyOT protocol introduced by Nielsen et al. in 2012 lead to inferior performance and disfavor the MiniMAC approach for this work [30].

Reverse-Multiplication-Friendly Embeddings. The 2018 work by Cascudo et al. [11] investigates perfect security with an honest supermajority against an active adversary in the binary circuit setting. The work is unique because the packing performed uses a different trade-off than the classical packing, allowing the corruption threshold to stay intact. This trade-off is in the dimension of the extension field used for secret shares. As individual parties' secret shares must be unique to retain security in MPC, extension fields are used. These extension field values map onto the actual values used in the computation while providing uniqueness for the shares. While this can be seen as a weakness, the authors leverage precisely this blow-up in field size to convey additional information for parallel circuit execution. Also originally intended for pure SIMD-parallelized execution of the same circuit, it has since been adapted to non-SIMD circuits [20]. Our implementation will follow this approach.

4.2 Parameter Selection

Here, we are going to evaluate and lay out which RMFE parameters are suitable for Ascon evaluation in MPC. As mentioned earlier, several practical RMFE constructions are possible, and hence they need to be evaluated here.

At first glance, a maximal packing may seem optimal. Considering the evaluation of a single Ascon instance, the most intuitive options are a packing of 320 bits and thus the whole state, or 64 bits and thus a state word into a one share. First, utilizing the 64-bit packings instead of 320-bit packings reduces local computation cost as fewer deconstruction and re-ordering operations are necessary. When using the larger packing, individual words have to first be extracted from the packing before word-wise operations can be computed, e.g., for the S-box. Second, 64-bit packings allow for neat evaluation of the circuit without too many deviations from Ascon control flow, since the packings can represent exactly one word. For 64 bits to be packed together, a 66-bit packing is the smallest to be constructed using the proposed practical constructions from Cascudo et al.'s 2018 paper. The corresponding $(66, 215)_2$-RMFE is a $(3r, 10r - 5)_2$-RMFE, with $r = 22$ and is constructed by concatenation of a $(3, 5)_2$-RMFE and a $(22, 43)_{32}$-RMFE. The leftover bits would not be used.

While an Ascon evaluation using this packing will reduce the number of transmitted field elements in the substitution layer drastically, it maps into $\mathbb{F}_{2^{215}}$. Specifically, for the substitution layer, the bigger the mapping, the fewer field elements will be transmitted. However, the trade-off between packing size and required field size only makes sense if an appropriate number of parties is present that already require a similarly sized extension field for security. The implication is that if we are considering a realistic number of protocol participants n, the $(66, 215)_2$-RMFE maps into way bigger fields than those a plain protocol like BH08 would map into – BH08 requires extension field size $|\mathbb{F}| > 2n$. Thus, every advantage in field elements is likely lost in the number of bits transmitted when a factor of 215 is applied rather than a factor realistic in the realm of small to large scale MPC ($3 - 9$; at most 16 if considering massive scale MPC comparable to the number of full nodes present in the Bitcoin network[3]). However, it should be noted that RMFEs were designed to leverage existing extension fields. Following this, in the case of more realistic numbers of parties, smaller RMFEs are likely to outperform a plain protocol.

Thus, rather than using these big RMFEs, we will utilize smaller RMFEs, specifically $(3, 5)_2$ and $(4, 9)_2$ going forward, as these are the smallest RMFEs per Cascudo et al.'s 2018 paper. More details on the range of appropriate t, n will be discussed in Sect. 4.4.

4.3 Protocol: Online Phase

The SIMD adjusted Ascon evaluation will be described here. Together with the protocols described in Sect. 2.3, these form an RMFE-based online protocol that evaluates Ascon in MPC.

We detail the bit-rotation protocol in Fig. 5 and the MPC operations for constant, substitution and linear layer in Fig. 6. Each state word is packed into $\ell = \lceil 64/k \rceil$ field elements that each pack k bits. If k does not divide 64, the remaining bits are set to0. The control flow of the Ascon decryption is kept largely the same with the operations of the Ascon decryption translated into MPC operations on field elements that together form packed state words.

For the Ascon round function, the constant layer simply adds the RMFE-mapped round constant to the appropriate field element. The substitution layer computes the Boolean circuit S-box for the i th element of each state word. Finally, the linear layer uses the bit-rotation protocol (Fig. 5), that internally uses the modified `LinearTrans` protocol from Fig. 4 on specially chosen linear functions L_1 and L_2.

4.4 Cost Evaluation

In this section, we first analyze the costs associated with the previously specified online protocol. Next, we assess the cost of implementing Ascon within this framework and compare it against a chosen baseline.

For our baseline, we adopt the protocol by Beerliová-Trubíniová and Hirt [8], hereafter referred to as BH08. This choice is motivated by its recognition as the leading protocol for our security setting, as highlighted in the work of Cascudo et al. [11].

4.4.1 Baseline Costs. In this section, we briefly describe how Ascon is evaluated using the BH08 protocol to analyze the communication costs. The protocol operates over the base field $\mathbb{F}_{2^5}$ as specified in BH08, with exemplary parameters $t = 4$ and $n = 13$, corresponding to the largest number of parties present with $n = 13$. Further performance numbers are evaluated later.

Since no packing is applied, the previously defined 64-bit operations can be understood as being performed on individual bits. However, because this protocol supports parallel multiplication,

[3] Currently reachable Bitcoin nodes according to Bitnodes, 22,840 at the time of writing.

MultiRightRotate

The protocol is parametrized by the RMFE maps $\phi : \mathbb{F}_2^k \to \mathbb{F}_{2^m}$ and $\psi : \mathbb{F}_{2^m} \to \mathbb{F}_2^k$.
Input: $[\phi(\vec{x}_1)], \ldots, [\phi(\vec{x}_M)]$ input sharings, set J of rotation offsets.
Output: $\left[\phi(\vec{y}_p)\right]$ for all $p = 1, \ldots, M$, where $\vec{y}_1 || \ldots || \vec{y}_M = \sum_{j \in J} L^{(j)}(\vec{x}_1 || \ldots || \vec{x}_M)$ and $L^{(j)}$ is a cyclic right rotation by j positions.

(1) For each $j \in J$ with $k \mid j$, reorder the inputs as
$$\left[\phi\left(\vec{z}^{(j)}_{p+j/k \bmod M}\right)\right] := \left[\phi(\vec{x}_p)\right] .$$
(2) For each $j \in J$ with $k \nmid j$, all parties take fresh correlated random sharings from preprocessing:
$$\left\{\left[\phi(\vec{r}_p)\right], \left[\phi\left(L_1^{(j)}(\vec{r}_p)\right)\right], \left[\phi\left(L_2^{(j)}(\vec{r}_p)\right)\right]\right\}_{p=1\ldots M} .$$
with $L_1^{(j)}(x_1, \ldots, x_k) = (0, \ldots, 0, x_1, \ldots, x_{k-j})$ and $L_2^{(j)}(x_1, \ldots, x_k) = (x_{k-j+1}, \ldots, x_k, 0, \ldots)$.
(3) All parties call ReconsPubl on the M sharings $\left[\phi(\vec{x}_p)\right] + \left[\phi(\vec{r}_p)\right]$ for $p = 1, \ldots, M$, obtaining $\phi(\vec{x}_p) + \phi(\vec{r}_p)$.
(4) Each party applies $\psi(\cdot)$ locally to each revealed value to recover $\vec{x}'_p = \vec{x}_p + \vec{r}_p$.
(5) For each $j \in J$ with $k \nmid j$ and $p = 1 \ldots M$, calculate
$$\vec{z}^{(j)}_{p+\lfloor j/k \rfloor \bmod M} := L_1^{(j)}\left(\vec{x}'_p\right) + L_2^{(j)}\left(\vec{x}'_{p-1 \bmod M}\right) .$$
(6) Embed and unblind: For each $j \in J$ with $k \nmid j$ and $p = 1 \ldots M$:
$$\left[\phi\left(\vec{z}^{(j)}_{p+\lceil j/k \rceil \bmod M}\right)\right] := \phi\left(\vec{z}^{(j)}_p\right) + \left[\phi\left(L_1^{(j)}(\vec{r}_p)\right)\right] + \left[\phi\left(L_2^{(j)}\left(\vec{r}_{p-1 \bmod M}\right)\right)\right] .$$
(7) Sum and output: $\left[\phi(\vec{y}_p)\right] := \sum_{j \in J} \left[\phi\left(\vec{z}^{(j)}_p\right)\right]$, for $p = 1 \ldots M$.

Figure 5: RMFE-based SIMD right rotation protocol for any set $\mathcal{J}$ of rotation offsets on the same stateword.

the choice of t significantly impacts the communication complexity. Minimal extension field sizes are assumed, respecting BH08's requirement that $|\mathbb{F}| > 2n$, which for this setting implies $|\mathbb{F}| > 26$. Hence, the field is chosen as $\mathbb{F}_{2^5}$. Note that secret sharing during the input phase and processing in the output phase is not included here, as in a client-COED scenario the end-user, rather than the parties, performs this operation. The focus of the cost evaluation is on the Ascon permutation, which poses the protocol's sole communication overhead and encompasses all nonlinear operations, including AND gates and rotations.

Table 1 summarizes the communication costs of evaluating Ascon under these parameters. Linear operations such as p_L do not incur communication costs because they consist solely of XORs and reordering of bit-shares. The cost of evaluating p therefore depends on the cost of evaluating the nonlinear component p_S.

Ascon Round Function Operations

The protocol is parametrized by the RMFE maps $\phi : \mathbb{F}_2^k \to \mathbb{F}_{2^m}$. We let $\ell = \lceil 64/k \rceil$ denote the number of embeddings per word.
State: We represent the Ascon internal state of 320 bit as 5 vectors $\left[\vec{S}_0\right], \ldots, \left[\vec{S}_4\right]$ where $\left[\vec{S}_i\right] = \left(\left[\phi(\vec{x}^{(1)})\right], \ldots, \left[\phi(\vec{x}^{(\ell)})\right]\right)$ with the invariant that $\vec{x}^{(1)}$ until $\vec{x}^{(\ell-1)}$ together with the first $64 - \lfloor k/64 \rfloor \cdot k$ bits of $\vec{x}^{(\ell)}$ form the 64-bit state word.

Constant Layer p_C

$\left[\vec{S}_2\right] := \left[\vec{S}_2\right] + c_r$
c_r denotes the round constant of permutation round r.

Substitution Layer p_S

Each party P_i computes
$$\left(\left[\vec{S}_0\right], \ldots, \left[\vec{S}_4\right]\right) := \text{S-Box}\left(\left[\vec{S}_0\right], \ldots, \left[\vec{S}_4\right]\right),$$
according to Fig. 2 where

- $\left[\vec{S}_i\right] \oplus \left[\vec{S}_j\right]$ is defined as running addition component-wise.
- $\left[\vec{S}_i\right] \odot \left[\vec{S}_j\right]$ is defined as running the multiplication protocol in Fig. 3 component-wise.

Linear Layer p_L

Each party P_i makes use of MultiRightRotate (cf. Fig. 5) to compute:
$$\begin{aligned}
\left[\vec{S}_0\right] &:= \left[\vec{S}_0\right] + \text{MultiRightRotate}\left(\left[\vec{S}_0\right], \{19, 28\}\right), \\
\left[\vec{S}_1\right] &:= \left[\vec{S}_1\right] + \text{MultiRightRotate}\left(\left[\vec{S}_1\right], \{61, 39\}\right), \\
\left[\vec{S}_2\right] &:= \left[\vec{S}_2\right] + \text{MultiRightRotate}\left(\left[\vec{S}_2\right], \{1, 6\}\right), \\
\left[\vec{S}_3\right] &:= \left[\vec{S}_3\right] + \text{MultiRightRotate}\left(\left[\vec{S}_3\right], \{10, 17\}\right), \\
\left[\vec{S}_4\right] &:= \left[\vec{S}_4\right] + \text{MultiRightRotate}\left(\left[\vec{S}_4\right], \{7, 41\}\right) .
\end{aligned}$$

Figure 6: The Ascon round function operations expressed as RMFE-based computation.

The operation p_S corresponds to evaluating the five parallel 64-bit S-boxes per state, where each S-box comprises 320 parallel 1-bit AND gates. Consequently, the communication cost of p_S reflects this structure and appears as the first row in Table 1. The number of field elements transmitted by p_S can be estimated by
$$\left\lceil \frac{320}{\lfloor \frac{n-2t}{2} \rfloor} \right\rceil \cdot (2n^2 - 2n).$$
Here, the numerator 320 represents the total number of parallel 1-bit AND gates, which are parallelized up to $\lfloor (n-2t)/2 \rfloor$ multiplications per the BH08 protocol. The factor $2n^2 - 2n$ corresponds

Table 1: Costs of Ascon in BH08 [8] with $\mathbb{F}_{2^5}$, $t = 4$, $n = 13$.

Operation	Rounds	Field Elements	Bits
p_S	2	49,920	249,600
p_L	0	0	0
p	2	49,920	249,600
p^8	16	399,360	1,996,800
p^{12}	24	599,040	2,995,200
Init.+Fin.	48	1,198,080	5,990,400
64 128-bit blocks	1,024	25,559,040	127,795,200
1 KiB incl. Init+Fin	1,072	26,757,120	133,785,600

to the field element cost of each parallelized AND gate. For $t = 4$ and $n = 13$, this calculation yields $\left\lceil \frac{320}{\lfloor (13-8)/2 \rfloor} \right\rceil \cdot (2 \cdot 13^2 - 2 \cdot 13) = \left\lceil \frac{320}{2} \right\rceil \cdot 312 = 160 \cdot 312 = 49{,}920$ field elements. The communication cost in bits is obtained by multiplying this element count by the bit-size of each element, which is 5 bits in $\mathbb{F}_{2^5}$, resulting in $49{,}920 \times 5 = 249{,}600$ bits.

4.4.2 Protocol Costs. This section evaluates the communication complexity of the RMFE-based online protocol in terms of rounds and field elements, based on the work of Cascudo et al., Goyal et al., and Beerliová-Trubíniová and Hirt [8, 11, 23]. We analyze the costs associated with the input, addition, multiplication, bit reordering, and output components of the protocol. Table 2 provides an overview of the communication overhead for each operation. The cost analysis for additional sub-protocols, such as those used for reconstructing values, is deferred to Appendix A.3.1.

Input and Output Phases. In the input phase, vectors are transformed into scalars via RMFE, followed by arithmetic secret sharing. An additional correctness check is performed on the input values. For M RMFE-mapped inputs, ensuring correct encoding requires $\lceil M/(n-2t) \rceil$ batched public reconstructions. This yields a cost of $\lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$ field elements, communicated in two rounds. Arithmetic share generation further requires $2M(n-1)$ field elements in a single round. Thus, the total communication cost for the input phase is $2M(n-1) + \lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$ field elements over three rounds.

The output phase does not introduce additional overhead for the RMFE-mapping; only the reconstruction of output values incurs a communication cost. Since the output size depends strongly on the Ascon use-case, we only account for output reconstruction costs in specific applications. For example, in the COED scenario, Ascon's secret-shared outputs may be directly reused in further MPC steps. Nevertheless, whenever an RMFE-mapped vector is opened to a party, it triggers the cost of the `ReconsPriv` protocol.

Addition and Multiplication. Additions in this protocol do not incur communication. Multiplication requires an arithmetic product followed by re-encoding of the result. An arithmetic multiplication can be executed in parallel on up to $\lfloor (n-2t)/2 \rfloor$ gates, costing $2n^2 - 2n$ field elements communicated over two rounds. For M multiplication pairs, batched re-encoding incurs $\lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$ field elements in two rounds. Therefore, the total cost for multiplication is

$$(2n^2 - 2n) \cdot \left(\left\lceil \frac{M}{\lfloor (n-2t)/2 \rfloor} \right\rceil + \left\lceil \frac{M}{n-2t} \right\rceil \right)$$

Table 2: Online protocol costs.

Operation	Rounds	Field Elements
Input Phase	3	$M \cdot (n-1) + \lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$
Mult. of M packed values	4	$(2n^2 - 2n) \cdot \left(\left\lceil \frac{M}{\lfloor (n-2t)/2 \rfloor} \right\rceil + \left\lceil \frac{M}{(n-2t)} \right\rceil \right)$
ROTR of M packed values	2	$\lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$

Table 3: Communication costs of Ascon in the RMFE-based protocol with $(3,5)_2$, $t = 4$, $n = 13$.

Operation	Rounds	Field Elements	Bits
p_S	4	24,024	120,120
p_L	2	6,864	34,320
p	6	30,888	154,440
p^8	48	247,104	1,235,520
p^{12}	72	370,656	1,853,280
Init.+Fin.	144	741,312	3,706,560
64 128-bit blocks	3,072	15,814,656	79,073,280
1 KiB incl. Init+Fin	3,216	16,555,968	82,779,840

field elements, over four rounds in total. Note that additional reconstruction steps are required for the RMFE-mapped outputs following the arithmetic computation.

Bit Rotation. Bit rotations are implemented using the re-ordering protocol described in Sect. 4.3. Exploiting batch processing, $n - 2t$ packed values can be rotated simultaneously, resulting in a communication cost of $\lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$ field elements in two rounds.

4.4.3 Ascon in Protocol Costs. Table 3 details the communication cost of the Ascon protocol when instantiated with the $(3,5)_2$ RMFE, in the example case of $t = 4$, and $n = 13$. In this setting, SIMD-style operations are leveraged, particularly for ROTR and AND gates. Compared to the straightforward approach, evaluating p_L now involves 10 ROTR operations, each requiring re-encoding and benefiting from amortization over packed values. Multiple operations can be evaluated in parallel, enabling to amortize the $2n^2 - 2n$ field element cost. Crucially, this happens over packed values; thus parallely opened three-bit values can be opened in an amortized manner. The amount of ROTR Operations are determined as $\lceil 64/3 \rceil \cdot 5$, and the respective field element cost as $\lceil 220/(n-2t) \rceil \cdot (2n^2 - 2n)$.

The field element cost for p_S reflects both re-encoding and arithmetic AND operations. The total communication for a full permutation round p is obtained by summing the costs for p_S and p_L. To compute the total number of bits transmitted, the number of communicated field elements is multiplied by 5, corresponding to the field size specified by the $(3,5)_2$ RMFE.

4.4.4 Comparison. Figure 7 presents the communication cost of the Ascon permutation with varying numbers of parties and a maximum security threshold, comparing the RMFE-based protocol (parametrized by $(3,5)_2$ and $(4,9)_2$) to the baseline BH08 protocol. The figure clearly demonstrates that RMFE-based packing yields lower overall communication, especially as the number of parties increases. In Appendix A.3.2, we further give a plot detailing the communication costs with varying numbers of parties and the minimal security threshold. For small party counts (e.g., $t = 1$, $n = 4$),

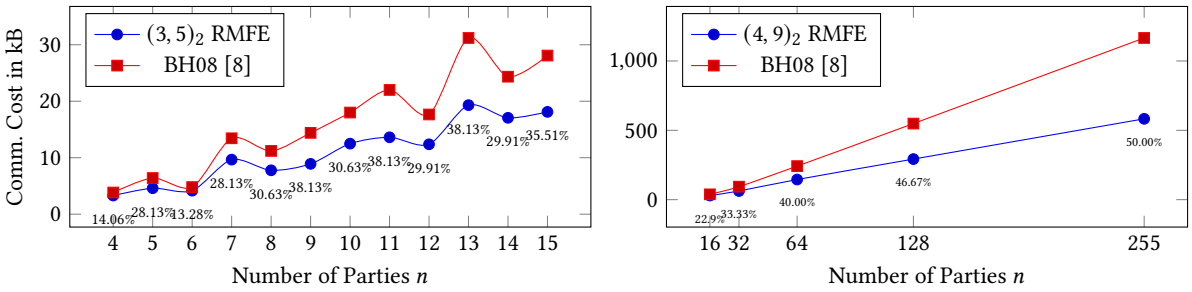


Figure 7: Comparison of online phase communication costs for one Ascon round function using $(3,5)_2$, $(4,9)_2$ RMFEs and BH08 [8]. The communication cost is shown for the maximal threshold $t = \lfloor n/3 \rfloor$ in kilobyte. The percentage indicates the improvement of our RMFE compared to BH08.

the RMFE-based protocol using $(3,5)_2$ transmits 26,400 bits per permutation, compared to 30,720 bits for the BH08 protocol. This represents a reduction of approximately 14%. With growing numbers of parties and threshold values, the communication advantage for RMFE becomes increasingly pronounced. At $t = 4$, $n = 13$, the RMFE protocol reduces the overall bits transmitted to 154,440, whereas the BH08 protocol requires 249,600 bits, a reduction of about 38%. For even larger settings, such as $t = 84$, $n = 255$, using an RMFE of type $(4,9)_2$, the total number of bits transmitted for one permutation drops to approximately 4.7 million in the RMFE protocol, compared to over 9.3 million with BH08, corresponding to a 50% improvement.

It should be noted that a lower security threshold results in a different performance improvement, in most cases lower than in the maximal threshold case. For example, a very large number of parties with a very low security threshold (e.g., $t = 1, n = 216, \ldots 255$) can result in the baseline performing on par with the RMFE-based protocol. However, this effect degrades quickly. In the maximal case of $n = 255$ considered, a $t > 20$ already yields a substantial improvement again. To better understand the non-linear performance improvement of the RMFE-based protocol versus the baseline, we list the contributing factors for performance roughly ordered by magnitude:

First, *extension field size*: RMFEs were designed to amortize extension fields already present. Hence, where an RMFE is chosen that maps into a bigger field than the baseline requires ($n = 4$, $(3,5)_2$), the advantage is smaller.

Second, *differing amortization effects in the substitution layer*: while the plain cost of the baseline and the RMFE-based protocol for performing the multiplication of scalar values itself is the same, the RMFE-based protocol additionally requires re-encoding. Furthermore, the amortization effects of these two parts of the RMFE-based evaluation are non-linear, as the former amortizes with $(n-2t)/2$ and the latter with $n-2t$. On top of this, the number of values required to be opened differs from the baseline to the RMFE-based protocol due to the packing (e.g., $22 \cdot 2$ vs $64 \cdot 2$ values opened during the arithmetic part of the multiplication).

Third, *linear layer cost and its amortization*: The linear layer first incurs cost because the packing needs to be reordered. Furthermore, it amortizes in a different manner to the baselines substitution layer since it is amortized with a factor of $n - 2t$. Thus, depending on how cleanly the number of values to be opened is divisible by $n-2t$ and $(n-2t)/2$, and the extension field size, heavily influences the characteristic of the performance improvement.

In summary, RMFE-based packing offers substantial communication reductions in realistic settings. These findings support the use of RMFE schemes for bandwidth-efficient secure evaluation of the Ascon permutation within multiparty computation.

5 Discussion

This paper systematically investigated approaches to optimize MPC evaluation of the Ascon permutation where the input is secret-shared. We identified SIMD-based approaches to be the most promising and give a concrete online protocol based on RMFEs. Perhaps counter-intuitively, we use relatively small RMFEs, i.e., packing only a few bits of a state word, in our protocol. This observation improves the communication cost of the Ascon permutation, which is the bottleneck in any MPC application that requires the use of Ascon.

Our MPC protocol is designed in the strong, information-theoretic setting and already outperforms the Boolean circuit-based BH08 baseline in the smallest evaluated configuration of $n = 4$ parties ($t = 1$), reducing communication by roughly 14%. At $n = 13$ parties ($t = 4$) the reduction grows to about 38%, and for large-party scenarios the savings reach up to 50%. (All figures are reported for the *maximum security threshold* admissible for each n, i.e. with t chosen as large as the setting allows.) These gains arise because the packing overhead diminishes rapidly while amortization over packed values becomes more effective as either the threshold or the number of parties increases. We also provided a Boolean circuit implementation of the Ascon-AEAD decryption in MP-SPDZ, which allows seamless transfer into various settings, including active security and dishonest majority.

Our results mainly improve the performance and bandwidth cost for transciphering scenarios where the use of Ascon is mandatory, e.g., due to Ascon being the NIST lightweight encryption standard. With our focus on the internal permutation, our results apply to all specified uses of Ascon: encryption, decryption, and hashing. At the same time, due to similar internal structure, we believe that our RMFE-based approach, including the bit-rotation protocol, will also improve the MPC evaluation of Keccak.

All our protocols make use of standard preprocessing, defined in [11]. Our bit-rotation protocols only require a specific correlation which is weaker than the generic network routing in [20]. Specialized protocols for this will likely outperform the generic preprocessing. We leave a description and thorough evaluation to future work.

Acknowledgments

This work was supported by CyberSecurity Research Flanders with reference number VR20192203, by the Flemish Government through FWO SBO project MOZAIK S003321N, and by the European Network of Excellence dAIEDGE under Grant Agreement Nr. 101120726.

References

[1] Aysajan Abidin, Enzo Marquet, Jerico Moeyersons, Xhulio Limani, Erik Pohle, Michiel Van Kenhove, Johann M. Marquez-Barja, Nina Slamnik-Krieštorac, and Bruno Volckaert. 2023. MOZAIK: An End-to-End Secure Data Sharing Platform. In *Proceedings of the Second ACM Data Economy Workshop (DEC '23)*. Association for Computing Machinery, New York, NY, USA, 34–40. doi:10.1145/3600046.3600052

[2] Aysajan Abidin, Erik Pohle, and Bart Preneel. 2024. Arithmetic Circuit Implementations of S-boxes for SKINNY and PHOTON in MPC. In *Computer Security – ESORICS 2023*, Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis (Eds.). Springer Nature Switzerland, Cham, 86–105. doi:10.1007/978-3-031-50594-2_5

[3] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology – ASIACRYPT 2016*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Vol. 10031. Springer Berlin Heidelberg, Berlin, Heidelberg, 191–219. doi:10.1007/978-3-662-53887-6_7

[4] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Vol. 9056. Springer Berlin Heidelberg, Berlin, Heidelberg, 430–454. doi:10.1007/978-3-662-46800-5_17

[5] Carsten Baum, Lennart Braun, Cyprien Delpech De Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. 2023. Publicly Verifiable Zero-Knowledge and Post-Quantum Signatures from VOLE-in-the-Head. In *Advances in Cryptology – CRYPTO 2023*, Helena Handschuh and Anna Lysyanskaya (Eds.). Vol. 14085. Springer Nature Switzerland, Cham, 581–615. doi:10.1007/978-3-031-38554-4_19

[6] Donald Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *Annual international cryptology conference*. Springer, 420–432.

[7] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. 2021. Order-C Secure Multiparty Computation for Highly Repetitive Circuits. In *Advances in Cryptology – EUROCRYPT 2021 (Lecture Notes in Computer Science)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer International Publishing, Cham, 663–693. doi:10.1007/978-3-030-77886-6_23

[8] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-Secure MPC with Linear Communication Complexity. In *Theory of Cryptography*, Ran Canetti (Ed.). Vol. 4948. Springer Berlin Heidelberg, Berlin, Heidelberg, 213–230. doi:10.1007/978-3-540-78524-8_13

[9] Amit Singh Bhati, Erik Pohle, Aysajan Abidin, Elena Andreeva, and Bart Preneel. 2023. Let's Go Eevee! A Friendly and Suitable Family of AEAD Modes for IoT-to-Cloud Secure Computation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Copenhagen Denmark, 2546–2560. doi:10.1145/3576915.3623091

[10] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2023. FLUTE: Fast and Secure Lookup Table Evaluations. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 515–533. doi:10.1109/SP46215.2023.10179345

[11] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. 2018. Amortized Complexity of Information-Theoretically Secure MPC Revisited. In *Advances in Cryptology – CRYPTO 2018 (Lecture Notes in Computer Science)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 395–426. doi:10.1007/978-3-319-96878-0_14

[12] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Benny Pinkas. 2018. High-Throughput Secure AES Computation. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '18)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3267973.3267977

[13] Ivan Damgård and Marcel Keller. 2010. Secure Multiparty AES. In *Financial Cryptography and Data Security*, Radu Sion (Ed.). Springer, Berlin, Heidelberg, 367–374. doi:10.1007/978-3-642-14577-3_31

[14] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *Advances in Cryptology - CRYPTO 2007 (Lecture Notes in Computer Science)*, Alfred Menezes (Ed.). Springer, Berlin, Heidelberg, 572–590. doi:10.1007/978-3-540-74143-5_32

[15] Ivan Damgård and Rasmus Zakarias. 2016. Fast Oblivious AES A Dedicated Application of the MiniMac Protocol. In *Progress in Cryptology – AFRICACRYPT 2016 (Lecture Notes in Computer Science)*, David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer International Publishing, Cham, 245–264. doi:10.1007/978-3-319-31517-1_13

[16] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P Smart. 2019. BBQ: Using AES in Picnic Signatures. In *International Conference on Selected Areas in Cryptography*. Springer, 669–692.

[17] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. 2018. Pushing the Communication Barrier in Secure Computation Using Lookup Tables.

[18] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2021. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *J Cryptol* 34, 3 (July 2021), 33. doi:10.1007/s00145-021-09398-9

[19] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. 2022. TurboPack: Honest Majority MPC with Constant Online Communication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 951–964. doi:10.1145/3548606.3560633

[20] Daniel Escudero, Yifan Song, and Wenhao Wang. 2025. Perfectly-Secure Multiparty Computation with Linear Communication Complexity over Any Modulus. In *Advances in Cryptology – ASIACRYPT 2024*, Kai-Min Chung and Yu Sasaki (Eds.). Vol. 15489. Springer Nature Singapore, Singapore, 102–132. doi:10.1007/978-981-96-0938-3_4

[21] David Evans, Vladimir Aleksandrovič Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation / David Evans, University of Virginia ; Vladimir Kolesnikov, Georgia Institute of Technology ; Mike Rosulek, Oregon State University. In *A Pragmatic Introduction to Secure Multi-Party Computation*. now Publishers Inc., Boston [u.a.

[22] Matthew Franklin and Moti Yung. 1992. Communication Complexity of Secure Computation. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*. 699–710.

[23] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. 2021. Unconditional Communication-Efficient MPC via Hall's Marriage Theorem. In *Advances in Cryptology – CRYPTO 2021 (Lecture Notes in Computer Science)*, Tal Malkin and Chris Peikert (Eds.). Springer International Publishing, Cham, 275–304. doi:10.1007/978-3-030-84245-1_10

[24] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-Knowledge from Secure Multiparty Computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*. ACM, San Diego California USA, 21–30. doi:10.1145/1250790.1250794

[25] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event USA, 1575–1590. doi:10.1145/3372297.3417872

[26] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. 2017. Faster Secure Multi-party Computation of AES and DES Using Lookup Tables. In *Applied Cryptography and Network Security (Lecture Notes in Computer Science)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.). Springer International Publishing, Cham, 229–249. doi:10.1007/978-3-319-61204-1_12

[27] Yehuda Lindell. 2021. Secure Multiparty Computation. *Commun. ACM* 64, 1 (Jan. 2021), 86–96. doi:10.1145/3387108

[28] Kalikinkar Mandal and Guang Gong. 2020. Can LWC and PEC Be Friends?: Evaluating Lightweight Ciphers in Privacy-Enhancing Cryptography. In *Fourth Lightweight Cryptography Workshop. NIST*.

[29] Hiraku Morita, Erik Pohle, Kunihiko Sadakane, Peter Scholl, Kazunari Tozawa, and Daniel Tschudi. 2025. MAESTRO: Multi-party AES Using Lookup Tables. In *USENIX Security 2025*. USENIX Association.

[30] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *Advances in Cryptology – CRYPTO 2012 (Lecture Notes in Computer Science)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Berlin, Heidelberg, 681–700. doi:10.1007/978-3-642-32009-5_40

[31] René Peralta and Luís T.A.N. Brandão. 2023. *NIST First Call for Multi-Party Threshold Schemes*. Technical Report NIST IR 8214C ipd. National Institute of Standards and Technology, Gaithersburg, MD. NIST IR 8214C ipd pages. doi:10.6028/NIST.IR.8214C.ipd

[32] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *Advances in Cryptology – ASIACRYPT 2009*, Mitsuru Matsui (Ed.). Springer, Berlin, Heidelberg, 250–267. doi:10.1007/978-3-642-10366-7_15

[33] Amit Sahai. 2008. Computing on Encrypted Data. In *Information Systems Security*, R. Sekar and Arun K. Pujari (Eds.). Vol. 5352. Springer Berlin Heidelberg, Berlin, Heidelberg, 148–153. doi:10.1007/978-3-540-89862-7_12

[34] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. doi:10.1145/359168.359176

[35] Meltem Sonmez Turan. 2024. *ASCON Family for Constrained Environments: Authenticated Encryption, Hash, and Extendable OutputFunctions*. Technical Report NIST SP 800-232 ipd. National Institute of Standards and Technology,

Gaithersburg, MD. NIST SP 800–232 ipd pages. doi:10.6028/NIST.SP.800-232.ipd
[36] Ko Stoffelen. 2016. Optimizing S-box Implementations for Several Criteria Using SAT Solvers. In *International Conference on Fast Software Encryption*. Springer, 140–160.
[37] Indranil Thakur, Angshuman Karmakar, Chaoyun Li, and Bart Preneel. 2025. A Survey on Transciphering and Symmetric Ciphers for Homomorphic Encryption.
[38] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Lawrence E Bassham, Jinkeon Kang, Noah D Waller, John M Kelsey, and Deukjo Hong. 2023. *Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process.* Technical Report NIST IR 8454. National Institute of Standards and Technology (U.S.), Gaithersburg, MD. NIST IR 8454 pages. doi:10.6028/NIST.IR.8454
[39] Fuxin Zhang and Zhenyu Huang. 2023. Optimizing S-box Implementations Using SAT Solvers: Revisited.

A Appendix

A.1 Additional subprotocols for the online protocol

Here, we will cover the ReconsPubl and the ReconsPriv protocol, which serve as the foundation for the online protocol. They are based on the 2018 work by Cascudo et al. and the 2007 work by Damgård and Nielsen [11, 14] .

A.2 ReconsPriv.

Figure 8 shows the ReconsPriv protocol adapted from Beerliová-Trubíniová and Hirt [8]. It works by having each player P_i send their share s_i to a player P_j, who tries to reconstruct the secret s using the polynomial created before.

ReconsPriv

Input: P_j, $[s]$
Ouput: s

(1) Each party P_i wit $i \in 1, ..., N$, transmits their share s_i of s to player P_j.
(2) P_j computes the secret $s = p(0)$ by evaluating the degree-t polynomial $p(\cdot)$ formed by at least $t + 1$ of the received shares.

Figure 8: ReconsPriv Protocol as per [8].

A.2.1 ReconsPubl. Figure 9 shows the ReconsPubl protocol. This protocol is used to reconstruct the public values of $n - 2t$ inputs $[a_1], [a_2], ..., [a_{n-2t}]$. While it has quadratic communication complexity in transmitted elements, this is amortized by working on those multiple inputs.

For the sake of simplicity of notation, puncturing, i.e. removing protocol will not be explicitly noted here as opposed to the two mentioned papers.

The method to reconstruct public values is based on the work by Damgård and Nielsen in their 2007 paper [14], as Cascudo et al. as well as Beerliová-Trubíniová and Hirt use the same method. Hence, the ReconsPubl described will be used for our protocol aswell.

To perform the reconstructions, $\beta_1, \ldots, \beta_n \in \mathbb{F}_{2^m}$ are taken as fixed public values agreed on before, multiplied with their input-shares and summed up. This computation is akin to the polynomial evaluation via Vandermonde matrices. Then, the parties distribute their shares to all parties so that they can reconstruct the values. Afterward, the parties use a Reed-Solomon error decoding algorithm to recover the values. The parties can recover the original values $a_1, a_2, \ldots, a_{n-2t}$ from these values. This layered approach is chosen to ensure robustness.

RMFE ReconsPubl

Input: $[x_1], [x_2], \ldots, [x_{n-2t}]$, fix $\beta_1, ..., \beta_n$.
Output: $x_1, x_2, \ldots, x_{n-2t}$ for all parties.

(1) Define $y_j := \sum_{i=1}^{n-2t} x_i \beta_j^i$. All parties perform a local computation of $[y_j] = \sum_{i=1}^{n-2t} [x_i]\, \beta_j^i$ for each j.
(2) For every party to receive the y_j values:
  (a) Perform private Reconstruction of each y_i using ReconsPriv towards P_i
  (b) P_i forwards y_i to P_j for every i, j.
(3) P_j employs error decoding algorithm for Reed-Solomon-codes to retrieve $x_1, \ldots, x_{n-2t}$ from the values $\widetilde{y}_1, \ldots, \widetilde{y}_n$ received in the preceding step, given that $\widetilde{y}_j \neq y_j$ for at most t values for each j.

Figure 9: RMFE ReconsPubl Protocol adapted from [11], as also used similarly in [8, 14, 20]

A.3 Protocol Cost Evaluation

In the following, a high level evaluation of the additional subprotocols' cost is given as well as a concrete cost plot at minimal corruption.

A.3.1 Additional subprotocols.

ReconsPubl. The ReconsPubl protocol reconstructs $n - 2t$ values to all parties. For this, $2n^2 - 2n$ field elements are communicated in two rounds.

For this, a value is reconstructed ReconsPriv to each party; thus, $n(n-1)$ field elements are communicated for this step. The reconstructed value is sent n-times of each party to all parties; thus, $n(n-1)$ field elements are communicated for this step. In total, $2n^2 - 2n$ field elements are communicated in two rounds.

Input Correctness and Re-Encodeing. The checking of the correctness of a single value sharing relies on one call to the ReconsPubl protocol. Similarly, the re-encoding a single value sharing relies on one call to the ReconsPubl protocol. Thus, they have the same $2n^2 - 2n$ field elements communicated in two rounds cost as the ReconsPubl protocol. It has to be noted, though, that instaces of input correctness checking and re-encoding are possible to be batched, as the ReconsPubl protocol can be used for $n - 2t$ sharings in two rounds.

A.3.2 Cost Plot at minimal corruption. In Plot 10, the performance of the baseline BH08 protocol and our RMFE-based protocol is plotted for a range of n.

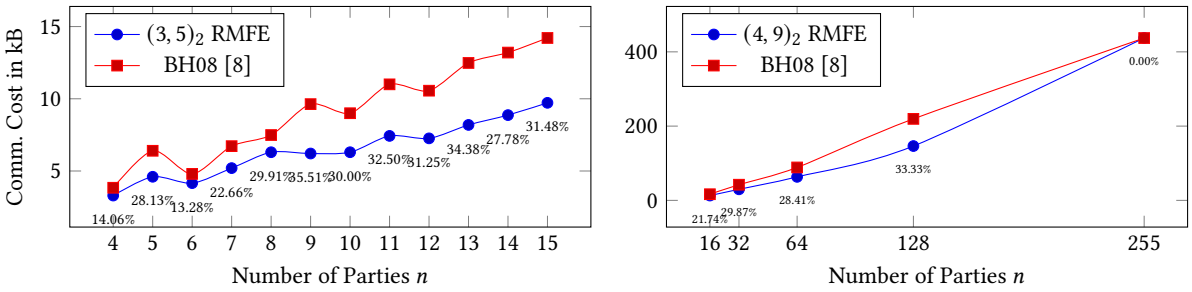


Figure 10: Comparison of online phase communication costs for one Ascon round function using $(3,5)_2$, $(4,9)_2$ RMFEs and BH08 [8] for minimal corruption ($t = 1$). The communication cost is shown in kilobyte, and the percentage indicates the improvement of our RMFE compared to BH08.

Operation	R	Field Elements
Arith. level Mult. of M values	2	$\lceil M/\lfloor (n-2t)/2 \rfloor \rceil \cdot 2n^2 - 2n$
ReconsPubl	2	$2n^2 - 2n$
M values ReEncode	2	$2n^2 - 2n \cdot \lceil M/(n-2t) \rceil$
Input Phase	3	$M \cdot (n-1) + \lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$
Mult. of M packed values	4	$(2n^2 - 2n) \cdot \left(\left\lceil \frac{M}{\lfloor (n-2t)/2 \rfloor} \right\rceil + \left\lceil \frac{M}{(n-2t)} \right\rceil \right)$
ROTR of M packed values	2	$\lceil M/(n-2t) \rceil \cdot (2n^2 - 2n)$

Table 4: Table of costs related to π with "R" meaning number of rounds necessary for the operation.

Table 5: Honest Majority, malicious security MP-SPDZ Benchmarking results of the naive Ascon implementation for 1 KiB of data including preprocessing, default MP-SPDZ settings.

Protocol	Parties	Rounds	Data sent	Time
ps-rep-ring.sh	3	800	2.68169	0.138682
mal-rep-ring.sh	3	800	2.68169	0.110613
mal-ccd.sh	3	2810	69.9723	18.5429

A.4 Baseline Implementation Benchmark Results

Table 5 details benchmark results of the MP-SDPZ implementation of Ascon.

A.5 MP-SPDZ Code

```
import copy
import os
import sys
from Compiler.types import Array, cint, regint
from Compiler.library import else_, for_range, foreach_enumerate,
    ↪if_, if_e, print_ln, print_str
from Compiler.compilerLib import Compiler
from Compiler.GC.types import sbits, cbits

compiler = Compiler()
debug = False
debugpermutation = False
debugrotr = False

def ascon_decrypt(key, nonce, associateddata, ciphertext, variant="
    ↪Ascon-128a"):
    """
    Ascon-128a decryption in MP-SPDZ.
    key: a bytes object of size 16 (2x64 bit, 128-bit security)
    nonce: a bytes object of size 16 (2x64 bit, must not repeat for
      ↪the same key!)
    associateddata: a bytes object of arbitrary length
    ciphertext: a bytes object of arbitrary length (also contains 2
      ↪x64-bit tag at end)
    variant: "Ascon-128a" (specifies key size, rate and number of
      ↪rounds)
    returns a bytes object containing the plaintext or None if
      ↪verification fails
    """
    sb64 = sbits.get_type(64)
    cb64 = cbits.get_type(64)
    S = [sb64(0), sb64(0), sb64(0), sb64(0), sb64(0)]
    k = 128 # bits
    a = 12   # rounds
    b = 8    # intermediate rounds for Ascon-128a
    rate_words = 2 # 128-bit blocks = 2 x 64-bit
    print_ln("Calling Ascon Init")
    ascon_initialize(S, k, rate_words, a, b, key, nonce)
    debugrotr = False
    print_ln("Calling Ascon Assoc")
    ascon_process_associated_data(S, b, rate_words, associateddata)
    print_ln("Calling Ascon Cipher")
    plaintext = ascon_process_ciphertext(S, b, rate_words,
      ↪ciphertext[:-2])
    print_ln("Calling Ascon Finalize")
    secret_tag = ascon_finalize(S, rate_words, a, key)
    ctagint_1 = secret_tag[0].reveal().to_regint()
    ctagint_2 = secret_tag[1].reveal().to_regint()
    tag_1 = ciphertext[-2].to_regint()
    tag_2 = ciphertext[-1].to_regint()
    plainplain = [0] * len(plaintext)
    @if_e(ctagint_1 == tag_1 )
    def _():
        @if_e(ctagint_2 == tag_2)
        def _():
            print_ln("No errors with the tags! Yay!")
        @else_
        def _():
            print_ln("Some Error with the second tag")
            print_ln("1: Computed: %s, Expected %s", ctagint_1,
      ↪tag_1)
            print_ln("2: Computed: %s, Expected %s", ctagint_2,
      ↪tag_2)
    @else_
    def _():
        print_ln("Some Error with the first tag")
        print_ln("1: Computed: %s, Expected %s", ctagint_1, tag_1)
        print_ln("2: Computed: %s, Expected %s", ctagint_2, tag_2)
    for index in range(len(plaintext)):
        plainplain[index] = plaintext[index].reveal()
    print_ln("Returning revealed plaintext")
    return plainplain
```

```python

# === Ascon AEAD building blocks ===
def ascon_initialize(S, k, rate_words, a, b, key, nonce):
    """
    Ascon initialization phase - internal helper function.
    S: Ascon state, a list of 5 sb64
    k: key size in bits
    rate_words: number of 64-bit words per rate (2 for Ascon-128a, 1 for Ascon-128)
    a: number of initialization/finalization rounds for permutation
    b: number of intermediate rounds for permutation
    key: a bytes object of size 16 (2x64-bit; 128-bit security)
    nonce: a bytes object of size 16
    returns nothing, updates S
    """
    sb64 = sbits.get_type(64)
    S[0] = sb64(0x80800c0800000000)  # IV for Ascon-128a (see spec)
    S[1] = key[0]
    S[2] = key[1]
    S[3] = sb64(nonce[0])
    S[4] = sb64(nonce[1])
    if debug: printstate(S, "initial value:")
    ascon_permutation(S, a)
    S[3] ^= key[0]
    S[4] ^= key[1]
    if debug: printstate(S, "initialization:")

def ascon_process_associated_data(S, b, rate_words, associateddata):
    """
    Ascon associated data processing phase - internal helper function.
    S: Ascon state, a list of 5 sb64
    b: number of intermediate rounds for permutation
    rate_words: number of 64-bit words per rate (2 for Ascon-128a)
    associateddata: a bytes object of arbitrary length
    returns nothing, updates S
    """
    sb64 = sbits.get_type(64)
    if not associateddata or len(associateddata) == 0:
        S[4] ^= sb64(1)
        if debug: printstate(S, "no associated data (finalize AD):")
        return
    ad_len = len(associateddata)
    for i in range(0, ad_len, 2):
        adblock0 = sb64(associateddata[i]) if i < ad_len else sb64(0)
        adblock1 = sb64(associateddata[i+1]) if (i+1) < ad_len else sb64(0)
        S[0] ^= adblock0
        S[1] ^= adblock1
        ascon_permutation(S, b)
    S[4] ^= sb64(1)
    if debug: printstate(S, "process associated data finalization:")

def ascon_process_ciphertext(S, b, rate_words, ciphertext):
    """
    Ascon ciphertext processing phase (during decryption) - internal helper function.
    S: Ascon state, a list of 5 sb64
    b: number of intermediate rounds for permutation
    rate_words: number of 64-bit words per rate (2 for Ascon-128a)
    ciphertext: a cb64 list of arbitrary length (length multiple of 2), tag is last two cb64s
    returns the plaintext, updates S
    """
    sb64 = sbits.get_type(64)
    plaintext = [0]*len(ciphertext)
    ct_len = len(ciphertext)
    # process two words (128 bits) per block
    for i in range(0, ct_len, 2):
        c0 = ciphertext[i] if i < ct_len else sb64(0)
        c1 = ciphertext[i+1] if (i+1) < ct_len else sb64(0)
        p0 = S[0] ^ c0
        p1 = S[1] ^ c1
        plaintext[i] = p0
        if (i + 1) < ct_len:
            plaintext[i + 1] = p1
        S[0] = c0
        S[1] = c1
        ascon_permutation(S, b)
    if debug: printstate(S, "process ciphertext:")
    return plaintext

def ascon_finalize(S, rate_words, a, key):
    """
    Ascon finalization phase - internal helper function.
    S: Ascon state, a list of 5 sb64
    rate_words: number of 64-bit words per rate (2 for Ascon-128a)
    a: number of initialization/finalization rounds for permutation
    key: a bytes object of size 16 (two 64-bit words)
    returns the tag, updates S
    """
    S[1] ^= key[0]
    S[2] ^= key[1]
    ascon_permutation(S, a)
    S[3] ^= key[0]
    S[4] ^= key[1]
    if debug: printstate(S, "finalization:")
    return [S[3],S[4]]

# === Ascon permutation ===
def ascon_permutation(S, rounds=1):
    """
    Ascon core permutation for the sponge construction - internal helper function.
    S: Ascon state, a list of 5 sb64
    rounds: number of rounds to perform
    returns nothing, updates S
    """
    if debugpermutation: printwords(S, "permutation input:")
    for r in range(12-rounds, 12):
        # --- add round constants ---
        S[2] ^= (0xf0 - r*0x10 + r*0x1)
        if debugpermutation: printwords(S, "round constant addition:")
        # --- substitution layer ---
        S[0] ^= S[4]
        S[4] ^= S[3]
        S[2] ^= S[1]
        sb64 = sbits.get_type(64)
        T = [sb64(0xFFFFFFFFFFFFFFFF),sb64(0xFFFFFFFFFFFFFFFF),sb64(0xFFFFFFFFFFFFFFFF),sb64(0xFFFFFFFFFFFFFFFF),sb64(0xFFFFFFFFFFFFFFFF)]
        for i in range(5):
            T[i] = (T[i]^S[i]) & S[(i+1)%5]
        for i in range(5):
            S[i] ^= T[(i+1)%5]
        S[1] ^= S[0]
        S[0] ^= S[4]
        S[3] ^= S[2]
        S[2] ^= 0xFFFFFFFFFFFFFFFF
        if debugpermutation: printwords(S, "substitution layer:")
        # --- linear diffusion layer ---
        S[0] ^= rotr(S[0], 19) ^ rotr(S[0], 28)
        debugrotr = False
        S[1] ^= rotr(S[1], 61) ^ rotr(S[1], 39)
        S[2] ^= rotr(S[2],  1) ^ rotr(S[2],  6)
        S[3] ^= rotr(S[3], 10) ^ rotr(S[3], 17)
```

```
        S[4] ^= rotr(S[4],  7) ^ rotr(S[4], 41)
        if debugpermutation: printwords(S, "linear diffusion layer:")

# === helper functions ===
def rotr(val, r):
    bits = val.bit_decompose()
    bits.reverse()
    l = len(val)
    left = bits[l-r-1:]
    right = bits[:l-r-1]
    comp = left + right
    comp.reverse()
    res = val.get_type(l).bit_compose(comp)
    return res

def printstate(S, description=""):
    print_ln(" State " + description)
    for i in range(len(S)):
        print_str('  S[%s]: %s; ',i, copy.copy(S[i]).reveal().to_regint())
    print_ln()

def printwords(S, description=""):
    print_ln(" " + description)
    for i in range(len(S)):
        print_str('  S[%s]: %s; ',i, copy.copy(S[i]).reveal().to_regint())
    print_ln()

def conv(x):
    return [int(x[i : i + 16], 16) for i in range(0, len(x), 16)]

def get_blocks(type, string):
    intarr = conv(string)
    typarr = type.Array(len(intarr))
    typarr.assign([type(x) for x in intarr])
    return typarr

def test_decryption():
    print_ln("################################")
    print_ln()
    test_cipher = 'f0f356233306bb4d054efb9338d7e237218f7f0447c9e4..'
    test_key = "904e853277fd4c0cf8d4e465cffa414b"
    test_nonce = '982be2236ec1bac0b8e69cd579ef0894'
    test_plain = 'abd5b55bfe0e8400c2a14f2048774b18ce539320319afeb..'
    cb64 = cbits.get_type(64)
    sb64 = sbits.get_type(64)
    ciphertext_blocks = [cb64(x) for x in conv(test_cipher)]
    print_ln("Ciphertext Input: ")
    for i in range(len(ciphertext_blocks)):
        print_str(" i: %s, val: %s;",i,ciphertext_blocks[i].to_regint())
    print_ln()
    print_ln()
    key = get_blocks(sb64,test_key)
    nonce = get_blocks(cb64,test_nonce)
    plain = [cb64(x) for x in conv(test_plain)]
    associated_data = []
    print_ln("Lets Go decrypt")
    print_ln()
    res = ascon_decrypt(key, nonce, associated_data, ciphertext_blocks)
    print_ln()
    print_ln('expected %s\n', plain)
    print_ln('got %s\n', res)

test_decryption()
```

Listing 1: Ascon-128a Decryption in MP-SPDZ