Coral: Maliciously Secure Computation Framework for Packed and Mixed Circuits

Zhicong Huang
Ant Group
Hangzhou, China
zhicong.hzc@antgroup.com

Wen-jie Lu*
Ant Group
Hangzhou, China
fionser@gmail.com

Yuchen Wang
Ant Group
Beijing, China
tianwu.wyc@antgroup.com

Cheng Hong†
Ant Group
Beijing, China
vince.hc@antgroup.com

Tao Wei
Ant Group
Hangzhou, China
lenx.wei@antgroup.com

WenGuang Chen
Ant Group
Beijing, China
yuanben.cwg@antgroup.com

Abstract

Achieving malicious security with high efficiency in dishonest-majority secure multiparty computation is a formidable challenge. The milestone works SPDZ and TinyOT have spawn a large family of protocols in this direction. For boolean circuits, state-of-the-art works (Cascudo et. al, TCC 2020 and Escudero et. al, CRYPTO 2022) have proposed schemes based on reverse multiplication-friendly embedding (RMFE) to reduce the amortized cost. However, these protocols are theoretically described and analyzed, resulting in a significant gap between theory and concrete efficiency.

Our work addresses existing gaps by refining and correcting several issues identified in prior research, leading to the first practically efficient realization of RMFE. We introduce an array of protocol enhancements, including RMFE-based quintuples and (extended) double-authenticated bits, aimed at improving the efficiency of maliciously secure boolean and mixed circuits. The culmination of these efforts is embodied in *Coral*, a comprehensive framework developed atop the MP-SPDZ library. Through rigorous evaluation across multiple benchmarks, Coral demonstrates a remarkable efficiency gain, outperforming the foremost theoretical approach by Escudero et al. (which incorporates our RMFE foundation albeit lacks our protocol enhancements) by a factor of 16-30×, and surpassing the leading practical implementation for Frederiksen et al. (ASIACRYPT 2015) by 4-7×.

CCS Concepts

• **Security and privacy → Information-theoretic techniques**.

Keywords

RMFE; MFE; embedding; pack; MPC; MAC; SPDZ; TinyOT; boolean; malicious; binary field; composite field; daBit; edaBit

*Also with Zhejiang University.
†Corresponding author.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690223

ACM Reference Format:
Zhicong Huang, Wen-jie Lu, Yuchen Wang, Cheng Hong, Tao Wei, and WenGuang Chen. 2024. *Coral*: Maliciously Secure Computation Framework for Packed and Mixed Circuits. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3658644.3690223

1 Introduction

Protocols for secure multiparty computation (MPC) enable a set of parties to jointly compute an agreed-upon function of their private inputs without revealing anything else other than the output. In recent years, MPC has rapidly moved from academic research to practical applications, thanks to various performant algorithmic upgrades and efficient implementations [26, 34, 35, 41].

In spite of these impressive results, maliciously secure MPC frameworks have been applied with limited scale in practice, due to their high performance penalty in communication and computation. Malicious security in the dishonest-majority setting is one of the strongest models in MPC, where more than half of the parties are corrupted by adversaries and may deviate arbitrarily from the prescribed protocol. This includes the interesting case of two-party computation. The SPDZ protocols [15, 16] fall into this setting over $\mathbb{Z}_q$ when q is a prime, and use *somewhat homomorphic encryption* for preprocessing (authenticating secret-shared inputs with information-theoretic MACs and generating multiplication triples). Follow-up works from Keller et al. (MASCOT [27] based on oblivious transfer, and Overdrive [28] based on RLWE homomorphic encryption) improve the performance by several orders of magnitude under the same setting. Frameworks for arithmetic circuits in the ring modulo $q = 2^k$ are also explored in this line of research (SPD$\mathbb{Z}_{2^k}$ [13] based on oblivious transfer, Overdrive2k [32] and MHz2k [12] based on RLWE homomorphic encryption).

To achieve the same security for boolean circuits, SPDZ-style or TinyOT-style protocols have been proposed to handle the $\mathbb{F}_2$ domain [7, 17, 21, 22, 29]. However, these protocols require the MAC on every secret-shared bit to be at least as big as the statistical security parameter (e.g., 40 bits), blowing up the complexity by a large factor. For example, the MP-SPDZ library [26] implements a Tinier protocol based on the work from Frederiksen et al. [21], appending a MAC in $\mathbb{F}_{2^{40}}$ to every shared bit. The SPD$\mathbb{Z}_{2^k}$ scheme could also

be adapted[1] to the special boolean case of $k = 1$, by sharing bits and MACs in $\mathbb{Z}_{2^{1+s}}$ for $s > 40$. To reduce the blown-up factor, a batch authentication idea has been applied in MiniMAC [17], by relying on a linear error correcting code (LECC) to combine a vector of bits into a codeword and authenticate them together with a single MAC. A similar LECC-based approach is also discussed in Committed MPC [22]. Recently, more efficient constructions based on *reverse multiplication-friendly embedding (RMFE)* are proposed [9, 20], achieving more compact encoding than the approach of LECC. In boolean circuits, RMFE maps from a vector space $\mathbb{F}_2^k$ to a field $\mathbb{F}_{2^m}$ ($m > k$), which essentially enables single-instruction-multiple-data (SIMD) computation in the vector space by performing field operations in the embedded field. Escudero et al. [20] also propose more efficient preprocessing protocols by embedding the field $\mathbb{F}_{2^m}$ into a larger vector space $\mathbb{F}_2^t$ ($t > m$) with *multiplication-friendly embedding (MFE)* such that oblivious linear evaluation (OLE) over $\mathbb{F}_{2^m}$ can be obtained by evaluating many small OLEs over $\mathbb{F}_2$. These RMFE-based MPC protocols give state-of-the-art performance over $\mathbb{F}_2$ as shown in their analysis. Comparing these works in terms of practical efficiency is challenging, as most lack both implementation and framework integration. Part of the contribution in this work is to further boost the concrete efficiency of state-of-the-art protocols with dedicated implementation.

For many applications, it generally gives optimal performance by mixing the evaluation of arithmetic circuits and boolean circuits. *Double-authenticated bit (daBit)* [38] is a general technique that authenticates the same bit in two domains, e.g., $\mathbb{Z}_q$ domain and $\mathbb{F}_2$ domain. It enables the design of many useful sub-routines such as comparison, equality testing and integer truncation, which are necessary to realize complex applications. However, for the above non-linear primitives that require binary circuits for intermediate computation, *extended double-authenticated bit (edaBit)* [19] has proven to be much more efficient. This technique authenticates an integer in $\mathbb{Z}_q$ and its bit decomposition in $\mathbb{F}_2$, which constitutes the randomness resource in a wide set of non-linear primitives. The MP-SPDZ library provides implementation of these techniques and allows convenient pairing between an arithmetic scheme and a boolean scheme.

1.1 Our Contributions

This work focuses on RMFE-based MPC, introducing novel primitives and applications for actively secure computation with a dishonest majority in *packed* (SIMD) and *mixed* (arithmetic and boolean gates) circuits. Recognizing the implementation gap in prior RMFE-based research between (R)MFE construction and its application in MPC, we present essential fixes, optimizations, and a highly efficient implementation to bridge this divide and operationalize (R)MFE. Leveraging these advancements, we design improved solutions for secure boolean computation and mixed-circuit computation, incorporating classical TinyOT insights and recent VOLE-style (*vector oblivious linear evaluation*) OT extension developments [6, 44]. Our contributions in this work are four-fold:

- We address the existing gaps within prior RMFE-based frameworks, present concrete instantiation, and provide highly efficient implementation of the underlying mathematics. This could be of general interest to the research line.
- We propose more efficient protocols for RMFE-based boolean circuit evaluation. Our protocols, crucial for input authentication and multiplication randomness generation, improve upon prior works' efficiency by at least one order of magnitude asymptotically.
- We construct RMFE-based daBit and edaBit generation based on a vectorization of previous protocols [19], achieving more efficient amortized RMFE-based mixed-circuit evaluation.
- We introduce *Coral*, a fully implemented framework in MP-SPDZ, along with compiler support for packed circuits. Extensive evaluation reveals *Coral* outperforms state-of-the-art methods by 4-30× in boolean circuits and 3-5× in mixed circuits across multiple efficiency metrics. **We open source** *Coral* **at**: https://github.com/AntCPLab/OpenCoral/.

1.2 Related Work

(R)MFE-based MPC. MFE in secure computation was first explored by Cascudo et al. [33], providing constructions we utilize herein. RMFE's application in this context was initiated by Cascudo et al. [8] and Block et al. [4]. The former employed RMFE to reduce communication overhead in honest-majority settings necessitating Shamir secret sharing over large finite fields. Conversely, the latter devised a protocol enhancing two-party evaluation of linear multiplication gates over a small field via a single multiplication over a larger field. Cascudo and Gundersen [9] constructed a complete RMFE-based framework for boolean circuits in the dishonest-majority setting, employing SPDZ-style MACs on the embedded field; however, it necessitated interactive re-encoding after each multiplication, incurring two communication rounds. Escudero et al. [20] improved these outcomes, introducing two key advancements for boolean circuits: replacing Beaver triples with quintuples to save a communication round and leveraging MFE for more efficient preprocessing. Furthermore, their work generalized to the $\mathbb{Z}_{p^k}$ domain, building upon recent advances in ring-based multiplicative secret sharing schemes [14].

SPDZ protocols. In the actively secure dishonest-majority setting, SPDZ (Damgård et al., [16]) has been a milestone work that spawns a family of protocols. It introduces the SPDZ-style MAC where a global MAC m for each value x and a global key α is shared among parties, satisfying the relation $m = \alpha x$. Besides the arithmetic protocols that are discussed in the previous paragraphs, recent works begin to seek further optimizations that target complex applications, such as matrix and convolution triples [11, 36]. These proposals are orthogonal to our work and can be paired with our scheme to boost concrete efficiency in applications. In fact, MiniMAC [17] and RMFE-based MPC [9, 20] belong to the SPDZ family as they also rely on the SPDZ-style MAC for authentication.

TinyOT protocols. Before SPDZ, information-theoretic MACs in arithmetic circuits have been studied by Bendlin et al. [3] in secret-sharing based multiparty computation that builds upon additive homomorphic encryption. These MACs are established between every pair of players and later referred to as BDOZ-style

[1] The preprocessing protocols are replaced, e.g., with [21] or [18].

MACs that are widely used in various TinyOT-style protocols for boolean circuits. TinyOT (Nielsen et al., [31]) introduces the usage of OT extension in the preprocessing phase of a two-party computation framework for boolean circuits. Several follow-up works extend TinyOT to the multi-party setting [7, 21, 29], but they instead produce SPDZ-style shares. Wang et al. [42] propose more efficient TinyOT-style preprocessing in a two-party scheme and later extend it to the multiparty setting [43]. These works give state-of-the-art performance for TinyOT triple generation. Up to now, techniques based on TinyOT remain the most efficient method for preprocessing AND triples in the dishonest-majority setting.

Mixed-circuit computation. [18] is a self-contained framework that extends SPD$\mathbb{Z}_{2^k}$ [13] and evaluates mixed circuits in $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_2$. It introduces the idea of obtaining $\mathbb{Z}_2$ triples by converting from TinyOT triples and performs daBit-style conversions between arithmetic sharing and boolean sharing. After the original introduction of daBit [38], later works have proposed more efficient daBit generation protocols in various secure computation contexts [1, 5, 37]. In addition to the contribution of edaBit, Escudero et al. [19] also propose efficient daBit construction in their work.

2 Preliminaries

2.1 Notations

For a set $\mathcal{D}$, $x \xleftarrow{\$} \mathcal{D}$ means x is sampled from $\mathcal{D}$ uniformly at random. The logical AND and XOR is $\wedge$ and $\oplus$, respectively. We use bold letters such as $\mathbf{a}$ to represent vectors, and use $\mathbf{a}[j]$ to denote the j-th component of $\mathbf{a}$. The Hadamard product of vectors is written as $\mathbf{a} \odot \mathbf{b}$.

2.2 Cryptographic Primitives

2.2.1 Authenticated Secret Sharing. Throughout this manuscript, we use additive secret sharing schemes. Malicious security is achieved by binding secret shares with information-theoretic MACs. Two prevalent styles of MACs are used in this work.

SPDZ-style MACs. In the SPDZ protocol family, for a secret value $x \in \mathbb{S}$, n parties maintain the authentication formula: $\sum_{i=1}^n x^i \cdot \sum_{i=1}^n \alpha^i = \sum_{i=1}^n m^i$, where party i holds value share x^i, MAC share m^i, and key share α^i, and $\alpha = \sum_{i=1}^n \alpha^i$ is a *global* secret MAC key. Let $[\![x]\!]$ denote SPDZ sharing where party i holds tuple (x^i, m^i), respectively (we omit α^i since it is the same across all sharings). The space $\mathbb{S}$ depends on concrete schemes. For example, in MASCOT [27], $\mathbb{S}$ is a prime field $\mathbb{F}_q$ or a binary field $\mathbb{F}_{2^\ell}$, whereas in Spdz2k [13], $\mathbb{S}$ is the ring $\mathbb{Z}_{2^\ell}$ (with shares x^i and m^i over a larger ring $\mathbb{Z}_{2^{\ell+s}}$). In this work, we use several different spaces (for both binary field and modulo ring) that will be highlighted whenever necessary.

BDOZ-style MACs. A *pairwise* MAC is used for boolean field $\mathbb{F}_2$ in various works including the TinyOT family of protocols [31, 42, 43]. For a secret bit $x \in \mathbb{F}_2$, party P_i holds a share x^i such that $x = \bigoplus_i x^i$. P_i authenticates its share to P_j, with P_i holding tuple $(x^i, M_j[x^i])$ and P_j holding tuple $(K_j[x^i], \Delta^j)$ such that: $M_j[x^i] \oplus K_j[x^i] = x^i \cdot \Delta^j$, where $M_j[x^i], K_j[x^i] \in \mathbb{F}_{2^\ell}$, and Δ^j is a fixed MAC key held by party j. Let $[\![x]\!]^B$ denote such TinyOT sharing that works as above.

2.2.2 Oblivious Transfer. We rely on oblivious transfer (OT) for several protocols in this work. In a general 1-out-of-2 OT, a sender inputs two messages m_0 and m_1 of length ℓ bits and a receiver inputs a choice bit $c \in \{0, 1\}$. At the end of the protocol, the receiver learns m_c, whereas the sender learns nothing. When sender messages are correlated, the Correlated OT (COT) is more efficient in communication [2]. In COT with XOR correlation, a sender inputs a function $f(x) = x \oplus \Delta$ for some $\Delta \in \mathbb{F}_{2^\ell}$, and a receiver inputs a choice bit c. At the end of the protocol, the sender learns $x \in \mathbb{F}_{2^\ell}$ whereas the receiver learns $x \oplus c \cdot \Delta \in \mathbb{F}_{2^\ell}$. OT can be implemented efficiently via IKNP-style OT extension [24] or VOLE-style OT extension [6, 44], and the latter proves to have a much lower communication when sender messages can be random.

2.2.3 (Reverse) Multiplication-Friendly Embedding. RMFE/MFE preserves multiplication (and addition trivially) between two spaces. In this work, we focus on their usage for improving boolean computation.

Definition 1 (RMFE ([8])). *A reverse multiplication-friendly embedding, or $(k, m)_q$-RMFE for short, is a pair of embedding map $\phi\colon \mathbb{F}_q^k \to \mathbb{F}_{q^m}$ and recovery map $\psi\colon \mathbb{F}_{q^m} \to \mathbb{F}_q^k$, such that $\forall \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^k$, it holds that $\mathbf{x} \odot \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$.*

In particular, $m \geq k$. We have the following results for constructing RMFE instances.

Theorem 1 (Lemma 4 in [8], and Theorem 2 in [9]). *For all $1 \leq k \leq q+1$, there exists a $(k, m)_q$-RMFE with $m = 2k-1$, and a $(k, m)_q$-RMFE with $m = 2k$.*

In the following, we describe the construction of maps for the case of $m = 2k$ in [9] (similarly for $m = 2k-1$). Let $\alpha \in \mathbb{F}_{q^m}$ such that $(1, \alpha, \cdots, \alpha^{m-1})$ is a basis of $\mathbb{F}_{q^m}$ as a $\mathbb{F}_q$-vector space. Let $\beta_1, \beta_2, \cdots, \beta_{k-1}$ be $k-1$ distinct elements in $\mathbb{F}_q$, and f_i be the coefficient of X^i in $f \in \mathbb{F}_q[X]_{\leq n}$ (polynomials whose degree is not greater than n and whose coefficients are in $\mathbb{F}_q$). We define maps:

$$\xi_1\colon \mathbb{F}_q[X]_{\leq k-1} \to \mathbb{F}_q^k; \qquad f \mapsto (f(\beta_1), \cdots, f(\beta_{k-1}), f_{k-1})$$
$$\pi_1\colon \mathbb{F}_q[X]_{\leq 2k-1} \to \mathbb{F}_{q^{2k}}; \quad f \mapsto f(\alpha)$$
$$\xi_2\colon \mathbb{F}_q[X]_{\leq 2k-1} \to \mathbb{F}_q^k; \qquad f \mapsto (f(\beta_1), \cdots, f(\beta_{k-1}), f_{2k-2})$$

The embedding and recovery maps are constructed as:

$$\phi\colon \pi_1 \circ \xi_1^{-1} \quad \text{and} \quad \psi\colon \xi_2 \circ \pi_1^{-1}$$

This construction satisfies the property $\mathbf{x} \odot \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$. In addition, we define the *normalization* map $\tau\colon \phi \circ \psi$. An element $x \in \mathbb{F}_{q^m}$ is called a *normal* element if $\tau(x) = x$. We abuse the notation ψ^{-1} to let $\psi^{-1}(\mathbf{x})$ denote the set of preimages for $\mathbf{x}$ under ψ.

Definition 2 (MFE ([33])). *A multiplication-friendly embedding, or $(t, m)_q$-MFE for short, is a pair of embedding map $\sigma\colon \mathbb{F}_{q^m} \to \mathbb{F}_q^t$ and recovery map $\rho\colon \mathbb{F}_q^t \to \mathbb{F}_{q^m}$, such that $\forall x, y \in \mathbb{F}_{q^m}$, it holds that $xy = \rho(\sigma(x) \odot \sigma(y))$.*

Similarly, $t \geq m$. We have the following results for constructing MFE instances:

Theorem 2 (Theorem 8 in [33]). *Let $m \geq 2$ be an integer with $q \geq 2m-2$, then there exists a $(t, m)_q$-MFE between $\mathbb{F}_{q^m}$ and $\mathbb{F}_q^t$ with $t = 2m-1$.*

To complete the above theorem, we provide a detailed construction of the two maps by adapting the result from [33]. Let $\beta_1, \beta_2, \cdots, \beta_{2m-2}$ be $2m-2$ distinct elements in $\mathbb{F}_q$. We define maps:

$$\mu_1 \colon \mathbb{F}_q[X]_{\le m-1} \to \mathbb{F}_{q^m}; \qquad f \mapsto f(\alpha)$$
$$\nu_1 \colon \mathbb{F}_q[X]_{\le m-1} \to \mathbb{F}_q^{2m-1};$$
$$f \mapsto (f(\beta_1), \cdots, f(\beta_{2m-2}), f_{m-1})$$
$$\mu_2 \colon \mathbb{F}_q[X]_{\le 2m-2} \to \mathbb{F}_q^{2m-1};$$
$$f \mapsto (f(\beta_1), \cdots, f(\beta_{2m-2}), f_{2m-2})$$
$$\nu_2 \colon \mathbb{F}_q[X]_{\le 2m-2} \to \mathbb{F}_{q^m}; \qquad f \mapsto f(\alpha)$$

Then, the embedding and recovery maps are constructed as:

$$\sigma \colon \nu_1 \circ \mu_1^{-1} \quad \text{and} \quad \rho \colon \nu_2 \circ \mu_2^{-1}$$

It can be shown that this construction satisfies the property $xy = \rho(\sigma(x) \odot \sigma(y))$, and we skip the proof here.

Concatenation [9]. To construct mappings for large space, it is necessary to concatenate small (R)MFE instances. A $(t_1, m_1)_q$-MFE and a $(t_2, m_2)_{q^{m_1}}$-MFE can be concatenated to produce a $(t_1 t_2, m_1 m_2)_q$-MFE. Similarly, a $(k_1, m_1)_q$-RMFE and a $(k_2, m_2)_{q^{m_1}}$-RMFE can be concatenated to produce a $(k_1 k_2, m_1 m_2)_q$-RMFE. For example, $(3, 5)_2$-RMFE and $(7, 13)_{32}$-RMFE give us $(21, 65)_2$-RMFE. There is still a large gap between the above results and its application in MPC, which we will address in Section 3.

2.2.4 Background of RMFE-based MPC. For boolean computation, our protocols build upon the previous best-performing RMFE-based MPC framework [20] that is proposed for the more general ring $\mathbb{Z}_{p^e}$. We restrict the setting to $p^e = 2$ and introduce necessary background only for this setting. The framework maintains a critical invariant throughout the whole circuit: the parties virtually store a vector $\mathbf{x} \in \mathbb{F}_2^k$ by actually storing an element $x \in \mathbb{F}_{2^m}$ with $\psi(x) = \mathbf{x}$. Online computation respects this invariant, accepting inputs from $\mathbb{F}_{2^m}$ and producing outputs in $\mathbb{F}_{2^m}$. We summarize their core protocols below:

Input. $[\![x]\!] \leftarrow \mathsf{RInput}(\mathbf{x}, P_i)$, where $\mathbf{x} \in \mathbb{F}_2^k$, and $x \in \mathbb{F}_{2^m}$. It secret shares and authenticates a private input vector $\mathbf{x}$ from party P_i.

Linear combination. $[\![z]\!] \leftarrow a[\![x]\!] + [\![y]\!] + b$, where $a \in \mathbb{F}_2$ and $b \in \mathbb{F}_{2^m}$. Because the recovery map ψ is only $\mathbb{F}_2$-linear, but not $\mathbb{F}_{2^m}$-linear, their constant multiplication only accepts a in $\mathbb{F}_2$. This might be sufficient for some SIMD evaluation, but we will show that it is not enough for simple circuits (e.g., comparison) and propose necessary extension.

Multiplication. $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$. A preprocessed quintuple $([\![a]\!], [\![b]\!], [\![\tau(a)]\!], [\![\tau(b)]\!], [\![\tau(a)\tau(b)]\!])$ is needed instead of a Beaver triple. In our work, a different type of quintuples is produced from a more efficient preprocessing phase.

Partial opening. $\mathbf{x} \leftarrow \mathsf{ROpen}([\![x]\!])$, where MAC checking can be deferred and batched for greater efficiency. A random authenticated kernel element $[\![r]\!]$ with $r \leftarrow \psi^{-1}(\mathbf{0})$ is necessary to mask $[\![x]\!]$ in order to avoid potential leakage.

Packing, Unpacking, and Repacking [20]. In this study, we adopt the information flow as presented in [20]. Initially, an input vector of k bits undergoes *packing* (RMFE encoding), resulting in an element within $\mathbb{F}_{2^m}$. This allows for the execution of operations on the packed element, mirroring the intended computations on the original bits via SIMD processing. Upon completion, we reverse the process by *unpacking* (RMFE decoding), extracting the length-k bit vector from any $\mathbb{F}_{2^m}$ output. Multiplication operations additionally require a *repacking* step, invoking the τ operator, which combines RMFE decoding and subsequent encoding.

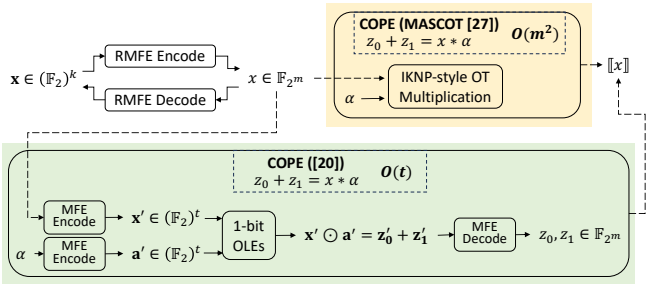


Figure 1: RMFE and MFE usage in [20]. RMFE is used to map between the input domain $\mathbb{F}_2^k$ and the computation domain $\mathbb{F}_{2^m}$, whereas MFE is used as an optimization for the offline preprocessing that authenticates an element in the computation domain. A comparison is given between the traditional MASCOT approach [27] and the MFE-based approach [20] for the authentication.

3 Bringing (R)MFE into Practice

In traditional SPDZ protocols, authenticating bits and enabling boolean computations requires appending an s-bit MAC to each bit. For instance, [21] uses $\mathbb{F}_{2^s}$ MACs for s-bit statistical security, while Spdz2k [13] represents bits with value and MAC shares in $\mathbb{Z}_{2^{1+s}}$ for $(s - \log s)$-bit security. This seemingly incurs a substantial time and space overhead, with s-fold increase. Recently, Cascudo et al.[9] and Escudero et al.[20] demonstrated the feasibility of packing k bits and authenticating them with a single MAC using (R)MFE. However, a significant gap exists between this theory and practical MPC applications. This section fills this gap, presenting a comprehensive framework integrating (R)MFE into boolean circuits.

We give a simplified overview of using (R)MFE for boolean circuits in Figure 1. To enable parallel computation for $\mathbf{x} \in \mathbb{F}_2^k$ (k bits), we first use RMFE to map $\mathbf{x}$ to $x \in \mathbb{F}_{2^m}$ (i.e., degree-m binary extension field). Afterwards, online computation works in the field $\mathbb{F}_{2^m}$. Traditionally, to authenticate P_0's input element $x \in \mathbb{F}_{2^m}$, MASCOT [27] uses an IKNP-style OT multiplication to compute an additive sharing of the product $x\alpha$, where α is P_1's MAC key. This stage essentially performs the *correlated oblivious product evaluation (COPE)*, where P_0 holds x, P_1 holds α, and they end up with z_0 and z_1 respectively such that $z_0 + z_1 = x\alpha$. The communication cost of this process is $O(m^2)$. To further boost this performance, Escudero et al. [20] propose to use MFE to map x (respectively, α) to $\mathbf{x}' \in \mathbb{F}_2^t$ (respectively, $\mathbf{a}' \in \mathbb{F}_2^t$) such that the communication cost drops from $O(m^2)$ to $O(t)$. Essentially, it computes a list of one-bit *oblivious linear evaluations (OLEs)*: $\mathbf{z_0'}[i] + \mathbf{z_1'}[i] = \mathbf{x}'[i] \cdot \mathbf{a}'[i]$, for $i \in \{1, \cdots, t\}$. Afterwards, an MFE decoding suffices to bring the resulting bits back to elements $z_0, z_1 \in \mathbb{F}_{2^m}$. Instead of directly working on $\mathbb{F}_{2^m}$, MFE allows to break down the COPE procedure

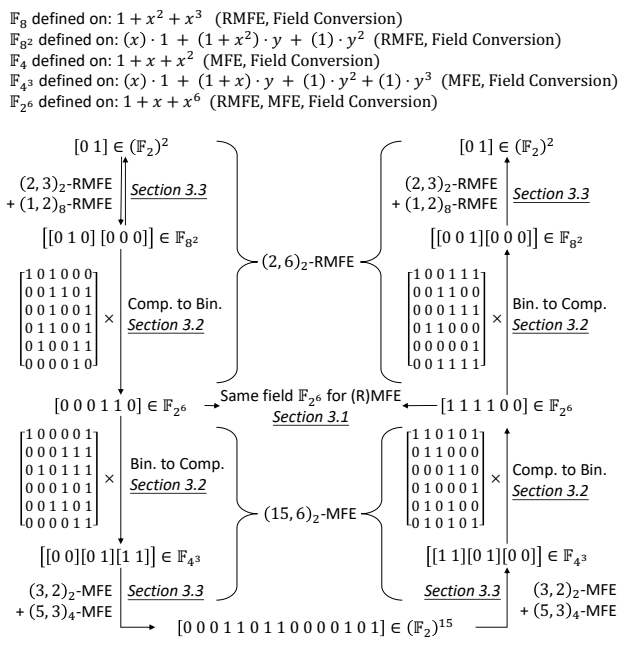


Figure 2: An example explaining the encoding and decoding procedures with our contributions. Various fields are defined on their respective irreducible polynomials. "Comp." is short for "Composite field" and "Bin." is short for "Binary field". The 6×6 conversion matrices are precomputed only once. Note: these example parameters are insecure for MPC due to their small size.

into t small OLEs over $\mathbb{F}_2$, offering enhanced execution efficiency. This approach also necessitates the field consistency between RMFE output and MFE input, which should both be $\mathbb{F}_{2^m}$. As introduced before, it is required that $k \leq m \leq t$.

We summarize our contribution in Figure 2 with an illustrative example. First of all, we fix a critical problem for choosing parameters so that RMFE and MFE can seamlessly connect with each other by defining them on the same binary field, e.g., $\mathbb{F}_{2^6}$ in Figure 2 (*Section 3.1*). Secondly, concatenating $(k_1, m_1)_2$-RMFE and $(k_2, m_2)_{2^{m_1}}$-RMFE only gives these conversions: $(\mathbb{F}_2)^{k_1 k_2} \leftrightarrow (\mathbb{F}_{2^{m_1}})^{k_2} \leftrightarrow \mathbb{F}_{(2^{m_1})^{m_2}}$. We fill in the last missing piece of composite/binary field conversion between $\mathbb{F}_{(2^{m_1})^{m_2}}$ and $\mathbb{F}_{2^{m_1 m_2}}$ so as to be interoperable with existing MPC frameworks (*Section 3.2*). Thirdly, some necessary revision is proposed to allow correct RMFE decoding, and to enable more efficient mapping computation (*Section 3.3*). At last, we analyze the implementation results with several engineering optimizations.

3.1 Bridging the Gap from RMFE to MFE

Figure 1 illustrates Escudero et al.'s [20] proposal of using MFE to minimize communication overhead for input authentication in the preprocessing phase. However, this approach is flawed if RMFE's output fails to align with MFE's input, a critical issue overlooked in their work, potentially impacting the scheme's overall efficiency. Specifically, certain constructions exhibit incompatibility between

Table 1: Example (R)MFE parameter sets. The 3 sets have similar expansion ratios and give at least 40-bit statistical security level in MPC. "concat." means a concatenated instance from the previous row. (R)MFEs on the left are obtained by concatenating those on the right, e.g., $(195, 42)_2$-MFE is obtained by concatenating $(3, 2)_2$-MFE, $(5, 3)_4$-MFE, and $(13, 7)_{64}$-MFE.

Example Param. 1 (RMFE ratio: 3, MFE ratio: 4.64)		
$(14, 42)_2$-RMFE	$(2, 3)_2$-RMFE	$(7, 14)_8$-RMFE
$(195, 42)_2$-MFE	$(3, 2)_2$-MFE *concat.*	$(5, 3)_4$-MFE $(13, 7)_{64}$-MFE
Example Param. 2 (RMFE ratio: 4, MFE ratio: 4.69)		
$(12, 48)_2$-RMFE	$(2, 4)_2$-RMFE	$(6, 12)_16$-RMFE
$(225, 48)_2$-MFE	$(3, 2)_2$-MFE *concat.*	$(5, 3)_4$-MFE $(15, 8)_{64}$-MFE
Example Param. 3 (RMFE ratio: 3, MFE ratio: 4.69)		
$(16, 48)_2$-RMFE	$(2, 3)_2$-RMFE	$(8, 16)_8$-RMFE
$(225, 48)_2$-MFE	$(3, 2)_2$-MFE *concat.*	$(5, 3)_4$-MFE $(15, 8)_{64}$-MFE

RMFE and MFE. For instance, the compact $(21, 65)_2$-RMFE (also employed in their complexity analysis) with an expansion ratio of $m/k = 65/21 \approx 3.1$ can be constructed by concatenating $(3, 5)_2$-RMFE and $(7, 13)_{32}$-RMFE. Nonetheless, the construction in Section 2.2.3 demonstrates the absence of a compatible MFE for this RMFE. Given our focus on binary computation, MFE concatenation must commence from a basic construction with $q = 2$. Theorem 2 implies that $m = 2$ when $m \geq 2$ and $q \geq 2m - 2$, resulting in all concatenated MFE with input from $\mathbb{F}_{2^{m'}}$ having even m' values due to multiplication with $m = 2$, rendering them incompatible with $\mathbb{F}_{2^{65}}$. Consequently, RMFE and MFE construction must be cohesively considered, a requirement neglected in prior studies.

Table 1 presents three practical and relatively efficient RMFE constructions. Set 1 offers the smallest expansion ratio 3 for $(14, 42)_2$-RMFE and 4.64 for $(195, 42)_2$-MFE. Set 2 (packing size 12) and Set 3 (packing size 16, a power of two) cater to applications favoring smaller or pow-of-two packing sizes, respectively.

3.2 Composite/Binary Field Conversion

Section 2.2.3 introduces concatenation to construct large mappings. However, concatenating $(k_1, m_1)_2$-RMFE and $(k_2, m_2)_{2^{m_1}}$-RMFE gives these conversions: $(\mathbb{F}_2)^{k_1 k_2} \leftrightarrow (\mathbb{F}_{2^{m_1}})^{k_2} \leftrightarrow \mathbb{F}_{(2^{m_1})^{m_2}}$. Mathematically speaking, $\mathbb{F}_{(2^{m_1})^{m_2}}$ is an equivalent field to $\mathbb{F}_{2^{m_1 m_2}}$ through isomorphism. From an implementation's point of view, in order to be interoperable with existing MPC frameworks (e.g., MP-SPDZ [26]) that are generally designed and implemented in $\mathbb{F}_{2^{m_1 m_2}}$, we need to explicitly compute this isomorphism to obtain elements in $\mathbb{F}_{2^{m_1 m_2}}$. Moreover, for later optimization with MFE, it is required that the input comes from the binary field $\mathbb{F}_{2^{m_1 m_2}}$, instead of a composite field $\mathbb{F}_{(2^{m_1})^{m_2}}$.

To fill the gap, we adopt the composite field representation construction method from [40]. It generates a conversion matrix $M \in \{0,1\}^{m \times m}$ that transforms $x' \in \mathbb{F}_{(2^{m_1})^{m_2}}$ into $x \in \mathbb{F}_{2^m}$

($m = m_1 m_2$) via $x = M \cdot x'$, where x' can be viewed as a simple concatenation of m_2 elements in $\mathbb{F}_{2^{m_1}}$. Denoting the transformation as θ, it preserves multiplication (and addition), i.e., $x'y' = \theta^{-1}(\theta(x')\theta(y'))$ for $x', y' \in \mathbb{F}_{(2^{m_1})^{m_2}}$. In Figure 2, $(2,3)_2$-RMFE and $(1,2)_8$-RMFE convert $[0,1] \in (\mathbb{F}_2)^2$ to $[[0,1,0],[0,0,0]] \in \mathbb{F}_{8^2}$ (viewed as a length-6 vector). Multiplying a precomputed matrix yields $[0,0,0,1,1,0] \in \mathbb{F}_{2^6}$.

To concretely instantiate the transformation, a degree-$(m_1 m_2)$ *primitive* polynomial is required to construct the binary field $\mathbb{F}_{2^{m_1 m_2}}$, and the various fields used in the whole encoding and decoding procedures have to be consistent, as shown in the top of Figure 2. Fortunately, we can use the primitive polynomials that have been found by Hansen and Mullen [23] back to the 90s.

3.3 Optimized Embedding/Recovery Maps

From embedding/recovery to encode/decode. Ideally, we want RMFE to serve as an encoding method so that we can store values and compute in the encoded domain $\mathbb{F}_{2^m}$, and decode back to the original domain $\mathbb{F}_2^k$ *correctly* when necessary. Namely, *decode* should be an inverse map of *encode*.[2] However, in the original definition of RMFE, this is not a necessary requirement. Actually, in its current construction in Section 2.2.3, the recovery map ψ is *not* an inverse function of the embedding map ϕ, i.e., $\psi(\phi(\mathbf{x})) \neq \mathbf{x}$ for some $\mathbf{x} \in \mathbb{F}_2^k$. Indeed, $\psi \circ \phi = \xi_2 \circ \pi_1^{-1} \circ \pi_1 \circ \xi_1^{-1} = \xi_2 \circ \xi_1^{-1}$, where ξ_1 and ξ_2 are not inverse of each other due to different arrangement for their images (f_{k-1} vs f_{2k-2}). Without the requirement of invertibility, **correctness is not guaranteed for some simple operations in MPC**, e.g., direct recovery of an input $\mathbf{x}$ or addition of two inputs immediately after embedding. In [20], they require that $\phi(\mathbf{1}) = 1$ in order to guarantee $\psi(\phi(\mathbf{x})) = \mathbf{x}$, without providing a concrete construction. One simple fix is to replace f_{k-1} and f_{2k-2} with evaluation on another distinct element, namely, $f(\beta_k)$ for $\beta_k \in \mathbb{F}_{2^m}$. Nonetheless, this requires interpolation and evaluation of polynomials with a higher degree. Instead, we propose to use f_0 for both. It can be proven that this construction maintains both the multiplicative property and $\phi(\mathbf{1}) = 1$. For both fixes, it is required in Theorem 1 that $k \leq q$ instead of $k \leq q+1$ because $\mathbb{F}_q$ needs to contain at least k distinct elements (f_0 is equivalent of evaluation on element 0).

Simplified basis construction. In practice, when constructing the embedding and recovery maps, we can select the basis in an efficient way. Let $P(X)$ be the irreducible polynomial used to construct field $\mathbb{F}_{q^m}$. Setting $\alpha = X \mod P(X)$ gives us the basis $(1, X, \cdots, X^{m-1})$. In this setting, μ_1 and π_1 turn out to be identity maps and thus can be removed. This gives the optimal computational performance of embedding and recovery, which happen frequently in almost every protocol (e.g., multiplication) and result in non-trivial runtime overhead in our observation.

To summarize, we obtain the following constructions in the end. Based on Section 2.2.3, we fix $\alpha = X \in \mathbb{F}_{q^m}$. MFE maps are constructed as: $\sigma = \nu_1$ and $\rho = \nu_2 \circ \mu_2^{-1}$, where we have abused the math notation of treating elements of $\mathbb{F}_{q^m}$ as those of $\mathbb{F}_q[X]_{\leq m-1}$ for the convenience of implementation. RMFE maps are constructed as:

$$\xi_1 : \mathbb{F}_q[X]_{\leq k-1} \to \mathbb{F}_q^k; \quad f \mapsto (f_0, f(\beta_1), \cdots, f(\beta_{k-1}))$$
$$\xi_2 : \mathbb{F}_q[X]_{\leq 2k-1} \to \mathbb{F}_q^k; \quad f \mapsto (f_0, f(\beta_1), \cdots, f(\beta_{k-1}))$$
$$\phi : \xi_1^{-1} \quad \text{and} \quad \psi : \xi_2$$

where $\beta_i \neq 0$ for all $i \in \{1, \cdots, k-1\}$.

[2] Since the two domains have different sizes, it is not required the other way around, i.e., encode does not need to be the inverse of decode.

Table 2: (R)MFE usage of different operations.

Op	MFE encode	MFE decode	RMFE encode	RMFE decode
Auth. k-bit Input	1	2	1	0
Quintuple	10	20	10	0
Secret Mult. (Online)	0	0	4	4

3.4 Optimizations and Implementation

Current RMFE-based MPC frameworks lack practical implementations, a challenge that can lead to (R)MFE computations becoming performance bottlenecks without adequate optimization. Considering the protocols in Section 4, Table 2 presents the RMFE and MFE usage for commonly employed boolean circuit operations. A straightforward sorting circuit for 128 32-bit integers, involving over 100 thousand AND gates (approx. 10 thousand quintuples), necessitates a highly efficient (R)MFE implementation to remain competitive with less CPU-intensive boolean MPC frameworks like [21].

Our baseline implementation, leveraging the NTL library [39], handles (R)MFE for fields and vectors of arbitrary length but falls short of the desired performance level. To enhance efficiency, we focus on the parameter sets in Table 1, complementing the optimized mappings from Section 3.3 with additional insights and optimizations that significantly improve performance.

Lookup table and cache. In MPC contexts, RMFE encoding can be implemented with lookup tables due to the restricted length of input vectors (e.g., 12 bits). This observation extends to encoding and decoding of numerous small instances of RMFE and MFE (e.g., $(15,6)_2$-MFE) that compose larger instances. For mappings exceeding lookup table capacity, a hash-based cache is utilized.

Small field precomputation. Mapping constructions rely heavily on operations (multiplication, power, inverse) in common small fields like $\mathbb{F}_{16}$ and $\mathbb{F}_{64}$. Consequently, these operations can be exhaustively precomputed and stored for subsequent use.

Figure 3 details the throughput of the baseline and optimized implementations for $(14,42)_2$-RMFE and $(195,42)_2$-MFE. The baseline, with its rate of several thousand operations per second, incurs a tens-of-seconds latency for tasks like sorting circuit computation, significantly lagging behind Tinier [21] (Table 7). Our optimizations elevate throughput to millions of operations per second, slashing computation time to under one second. Consequently, our protocols outperform state-of-the-art alternatives in both local and wide area network settings, as demonstrated in the evaluation.

4 Packed Circuit Optimization

This section presents improved secure boolean computation frameworks surpassing the prior state-of-the-art [20]. Enhancements

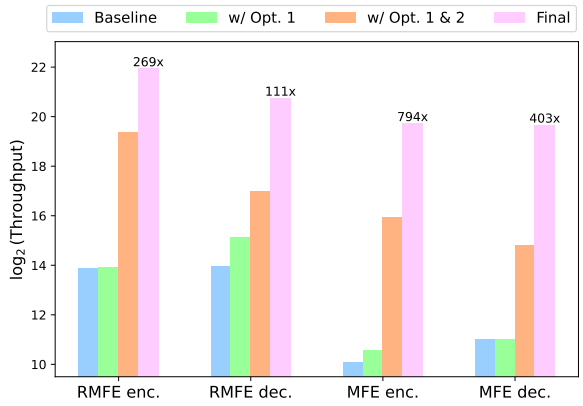


Figure 3: Throughput (operations per second) with various optimizations, for $(14, 42)_2$-RMFE and $(195, 42)_2$-MFE. 'Opt. 1' refers to optimized mapping in Section 3.3. 'Opt. 2' refers to lookup table and cache. 'Final' includes the previous two and small field precomputation. The improvement of 'Final' compared to 'Baseline' is annotated explicitly.

The functionality maintains a dictionary, TVal, to keep track of the authenticated values $[\![\cdot]\!]^B$ in TinyOT sharing, and RVal, to keep track of the authenticated values $[\![\cdot]\!]$ in RMFE sharing. We only list critical commands below, and leave the full specification of some standard commands in Appendix A, such as **tOpen**, **tCheck** for TinyOT sharing, and **rInput, rOpen, rCheck** for RMFE sharing.

tBit: On input (tBit, id) from all parties, sample a random value $a \in \mathbb{F}_2$, and set $\mathsf{TVal}[\mathsf{id}] \leftarrow a$.

tTriple: On input $(\mathsf{tTriple}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, sample two random values $a, b \in \mathbb{F}_2$, and set $(\mathsf{TVal}[\mathsf{id}_1], \mathsf{TVal}[\mathsf{id}_2], \mathsf{TVal}[\mathsf{id}_3]) \leftarrow (a, b, a \cdot b)$.

rQuintuple: On input $(\mathsf{rQuintuple}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3, \mathsf{id}_4, \mathsf{id}_5)$ from all parties, sample two random values $a, b \in \mathbb{F}_{2^m}$, and set $(\mathsf{RVal}[\mathsf{id}_1], \mathsf{RVal}[\mathsf{id}_2], \mathsf{RVal}[\mathsf{id}_3], , \mathsf{RVal}[\mathsf{id}_4], \mathsf{RVal}[\mathsf{id}_5]) \leftarrow (a, b, \phi(\psi(a) \odot \psi(b)), \tau(a), \tau(b))$.

rEnc: On input $(\mathsf{rEnc}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties, sample a random value $a \in \mathbb{F}_{2^m}$, and set $(\mathsf{RVal}[\mathsf{id}_1], \mathsf{RVal}[\mathsf{id}_2]) \leftarrow (a, \tau(a))$.

Figure 4: Functionality $\mathcal{F}_{\text{prep}}$

rConv: On input $(\mathsf{rConv}, \mathsf{tid}_1, \cdots, \mathsf{tid}_k, \mathsf{rid})$ from all parties, set $\mathsf{RVal}[\mathsf{rid}] \leftarrow \phi([\mathsf{TVal}[\mathsf{tid}_1], \cdots, \mathsf{TVal}[\mathsf{tid}_k]])$.

Figure 5: Functionality $\mathcal{F}_{\text{rConv}}$

include a more efficient preprocessing phase and essential extensions enabling comprehensive evaluation of typical circuits. An overview of the $\mathcal{F}_{\text{prep}}$ functionality employed in these protocols is provided in Figure 4.

Input: TinyOT sharings $[\![x_1]\!]^B, \cdots, [\![x_{n \cdot k}]\!]^B$
Output: RMFE sharings $[\![y_1]\!], \cdots, [\![y_n]\!]$, where $y_j = \phi([x_{(j-1)k+1}, \cdots, x_{jk}])$

[Construct]

1: Call $\mathcal{F}_{\text{prep.tBit}}$ to sample $s \cdot k$ additional shared bits $[\![r_1]\!]^B, \cdots, [\![r_{s \cdot k}]\!]^B$.
2: Each party P_i:
- partitions his shares (including those in step 1) into $(n + s)$ groups and computes:
$y_j^i = \phi([x^i_{(j-1)k+1}, \cdots, x^i_{jk}]), \forall j \in [1, n]$
$q_j^i = \phi([r^i_{(j-1)k+1}, \cdots, r^i_{jk}]), \forall j \in [1, s]$
- calls $\mathcal{F}_{\text{prep.rInput}}$ to obtain $[\![y_j^i]\!]$ and $[\![q_j^i]\!]$
3: Parties sum up the shares to obtain (possibly incorrect) $[\![y_j]\!] = \sum_i [\![y_j^i]\!]$, for $j \in [1, n]$, and $[\![q_j]\!] = \sum_i [\![q_j^i]\!]$, for $j \in [1, s]$.

[Check normality]

4: Sample $n \cdot s$ random bits $c_{j,g} \xleftarrow{\$} \{0, 1\}$, for $j \in [1, n], g \in [1, s]$.
5: Compute: $[\![w_g]\!] = (\sum_{j=1}^n c_{j,g} \cdot [\![y_j]\!]) + [\![q_g]\!]$ and call $\mathcal{F}_{\text{prep.rOpen}}$ to obtain w_g.
6: Check that $\tau(w_g) = w_g$ for all g. If not, abort.

[Check consistency]

7: Compute: $[\![z_{g,h}]\!]^B = (\sum_{j=1}^n c_{j,g} \cdot [\![x_{(j-1)k+h}]\!]^B) + [\![r_{(g-1)k+h}]\!]^B$ and call $\mathcal{F}_{\text{prep.tOpen}}$ to obtain $z_{g,h}$ for $h \in [1, k]$.
8: Check that $\psi(w_g) = [z_{g,1}, \cdots, z_{g,k}]$ for all g. If not, abort.
9: Call $\mathcal{F}_{\text{prep.rCheck}}$ and $\mathcal{F}_{\text{prep.tCheck}}$ on the opened values.
10: Output the sharings $[\![y_1]\!], \cdots, [\![y_n]\!]$

Figure 6: Protocol for converting TinyOT sharings to RMFE sharings Π_{rConv}.

4.1 General Boolean Share Conversion

Drawing inspiration from Damgård et al. [18], we observe that certain preprocessing materials, like AND triples, can be efficiently produced by adapting protocols from a source domain to a target domain, particularly when input authentication costs are low, as in our setting. Given authenticated boolean sharings $(\langle x_1 \rangle, \cdots, \langle x_k \rangle)$ with $x_i \in \mathbb{F}_2$ and $\langle \cdot \rangle$ representing a specific scheme, Figure 5 and Figure 6 outline the functionality and a general protocol for converting any valid authenticated boolean sharing (with $\langle \cdot \rangle$ instantiated as TinyOT sharing $[\![\cdot]\!]^B$) to RMFE sharing. This conversion proves instrumental in generating preprocessing materials more efficiently, exemplified by enhanced quintuple generation in Section 4.2.

Theorem 3. *Π_{rConv} securely implements $\mathcal{F}_{\text{rConv}}$ in the $\mathcal{F}_{\text{prep}}$-hybrid (Figure 4) model with statistical security parameter s.*

Proof (Sketch). $\mathcal{F}_{\text{rConv}}$ requires the conversion output to be both consistent and normal (note that $\tau(\phi(\mathbf{x})) = \phi(\mathbf{x})$). This is achieved by a normality check and a consistency check. Each check contains opening and checking s random linear combinations. If the adversary causes incorrect values to be authenticated as outputs, it can pass the check for each linear combination with a probability of at most 1/2. Detailed proof is provided in Appendix B. □

4.2 Quintuple Generation

To remove one online communication round from [9], Escudero et al. [20] propose to replace a triple with a quintuple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket \tau(a) \rrbracket, \llbracket \tau(b) \rrbracket, \llbracket \tau(a)\tau(b) \rrbracket)$, where $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$ are used for masking and the rest are used to maintain the invariant $\psi(z) = \mathbf{z}$ for the multiplication result $z \in \mathbb{F}_{2^m}$ and $\mathbf{z} = \mathbf{x} \odot \mathbf{y} \in \mathbb{F}_2^k$. In $\mathcal{F}_{\text{prep.rQuintuple}}$, our specification is actually different from that in the previous work, where they generate a quintuple by first generating random encoding pairs $(\llbracket a \rrbracket, \llbracket \tau(a) \rrbracket)$ and $(\llbracket b \rrbracket, \llbracket \tau(b) \rrbracket)$ with $\mathcal{F}_{\text{prep.rEnc}}$ and later applying a relatively expensive multiplication procedure to produce $\llbracket \tau(a)\tau(b) \rrbracket$. Their method applies a classical white-box OT-style construction to compute secret sharing of the product $\tau(a)\tau(b)$, and proceeds as in [22] and [9] with cut-and-choose, bucket sacrifice and bucket combine (twice) in order to prevent leakage of $\tau(a)$ and $\tau(b)$ in the OT-based multiplication. This has a cubic complexity $O(B^3)$ with a bucket size of B, which could be more costly than other existing bucketing protocols with quadratic or linear complexity [21, 42, 43]. To improve the quintuple generation, we prefer to first generate boolean triples with the efficient TinyOT approach, convert them to RMFE sharings with $\mathcal{F}_{\text{rConv}}$, and finally use standard sacrificing technique to complete the quintuples. Nonetheless, the quintuples generated by our protocol have a different presentation $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket, \llbracket \tau(a) \rrbracket, \llbracket \tau(b) \rrbracket)$ where $c = \phi(\psi(a)\psi(b))$, but we will later present a corresponding updated online multiplication protocol that relies on such quintuples.

Figure 7 gives our quintuple generation protocol. $\mathcal{F}_{\text{prep.tTriple}}$ and $\mathcal{F}_{\text{rConv}}$ guarantee that a generated triple $(\llbracket \hat{a}_j \rrbracket, \llbracket \hat{b}_j \rrbracket, \llbracket \hat{c}_j \rrbracket)$ has normality constraint ($\tau(\hat{a}_j) = \hat{a}_j$, etc.) and multiplication relation ($\hat{c}_j = \phi(\psi(\hat{a}_j) \odot \psi(\hat{b}_j))$). We still need to prepend the triple with random $a, b \in \mathbb{F}_{2^m}$ such that $\tau(a) = \hat{a}_j$ and $\tau(b) = \hat{b}_j$. To achieve this, a standard approach is applied: parties input satisfying elements, additively combine them, and finally sacrifice a small set of sharings. The sacrificing technique takes insight from the method Escudero et al. [20] use in generating encoding pairs $(\llbracket a \rrbracket, \llbracket \tau(a) \rrbracket)$ and kernel elements ($\llbracket r \rrbracket$ with $r \in \psi^{-1}(\mathbf{0})$), hence the security analysis flows through similarly.

Theorem 4. *$\Pi_{\text{rQuintuple}}$ securely implements $\mathcal{F}_{\text{prep.rQuintuple}}$ in the $(\mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{rConv}})$-hybrid model with s-bit statistical security.*

Proof (Sketch). $\mathcal{F}_{\text{rConv}}$ produces consistent and normal triples $(\llbracket \hat{a}_j \rrbracket, \llbracket \hat{b}_j \rrbracket, \llbracket \hat{c}_j \rrbracket)$. The remaining part of the protocol is to generate random a_j and b_j such that $\tau(a_j) = \hat{a}_j$ and $\tau(b_j) = \hat{b}_j$. This constraint is guaranteed by sacrificing additional s values through random linear combinations. Full proof is provided in Appendix B. □

The corresponding online multiplication protocol with our quintuple is presented in Figure 8. We have the following theorem with corresponding proof in Appendix B.

Theorem 5. *Π_{rMult} securely implements $\mathcal{F}_{\text{mpc.rMult}}$ in the $\mathcal{F}_{\text{prep}}$-hybrid model. (The standard online MPC functionality $\mathcal{F}_{\text{mpc}}$ is listed in Appendix A.)*

4.3 Extensions and Optimizations

4.3.1 Mixed-circuit computation. Prior RMFE-based MPC frameworks focus on either boolean circuits [9], or arithmetic circuits [14, 20]. Our work extends state-of-the-art ideas to facilitate mixed-circuit computation by incorporating RMFE-based *daBit* (double-authenticated bit [38]) and *edaBit* (extended double-authenticated bit [19]). DaBits, ubiquitous in mixed-circuit frameworks before edaBit's introduction, are integral to operations like Spdz2k comparison that involves arithmetic random bits and their boolean conversions (essentially a series of daBits). While edaBits offer efficiency gains in numerous mixed-circuit protocols (e.g., comparison),

Output: N quintuples $(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket, \llbracket c_i \rrbracket, \llbracket \tau(a_i) \rrbracket, \llbracket \tau(b_i) \rrbracket)$, for $i = 1, \cdots, N$, where $a_i, b_i \in R$ are random elements, and $\psi(c_i) = \psi(a_i) \odot \psi(b_i)$.

[Construct]

1: Parties call $\mathcal{F}_{\text{prep.tTriple}}$ to generate $(s+N) \cdot k$ TinyOT triples $(\llbracket x_i \rrbracket^B, \llbracket y_i \rrbracket^B, \llbracket z_i \rrbracket^B)$, for $i = 1, \cdots, (s+N) \cdot k$.

2: Parties call $\mathcal{F}_{\text{rConv}}$ to convert the above TinyOT sharings to RMFE sharings $(\llbracket \hat{a}_j \rrbracket, \llbracket \hat{b}_j \rrbracket, \llbracket \hat{c}_j \rrbracket)$, where for $j = 1, \cdots, s+N$,
$\hat{a}_j = \phi([x_{(j-1)k+1}, \cdots, x_{jk}])$,
$\hat{b}_j = \phi([y_{(j-1)k+1}, \cdots, y_{jk}])$,
$\hat{c}_j = \phi([z_{(j-1)k+1}, \cdots, z_{jk}])$.

3: P_i samples $a_j^i \xleftarrow{\$} \psi^{-1}([x^i_{(j-1)k+1}, \cdots, x^i_{jk}])$ and $b_j^i \in \psi^{-1}([y^i_{(j-1)k+1}, \cdots, y^i_{jk}])$ for $j = 1, \cdots, s+N$.

4: P_i call $\mathcal{F}_{\text{prep.rInput}}$ to obtain $\llbracket a_j^i \rrbracket$ and $\llbracket b_j^i \rrbracket$ for $j = 1, \cdots, s+N$.

5: Parties compute $\llbracket a_j \rrbracket = \sum_i \llbracket a_j^i \rrbracket$, and $\llbracket b_j \rrbracket = \sum_i \llbracket b_j^i \rrbracket$, for $j = 1, \cdots, s+N$.

[Sacrifice]

6: Parties sample random vectors $\mathbf{r}_h = (r_{h,1}, \cdots, r_{h,N}) \in \{0,1\}^N$ for $h \in [1, s]$.

7: Compute:
$\llbracket d_h \rrbracket = \sum_{j=1}^N r_{h,j} \llbracket \hat{a}_j \rrbracket + \llbracket \hat{a}_{N+h} \rrbracket$
$\llbracket e_h \rrbracket = \sum_{j=1}^N r_{h,j} \llbracket a_j \rrbracket + \llbracket a_{N+h} \rrbracket$
$\llbracket f_h \rrbracket = \sum_{j=1}^N r_{h,j} \llbracket \hat{b}_j \rrbracket + \llbracket \hat{b}_{N+h} \rrbracket$
$\llbracket g_h \rrbracket = \sum_{j=1}^N r_{h,j} \llbracket b_j \rrbracket + \llbracket b_{N+h} \rrbracket$
and call $\mathcal{F}_{\text{prep.rOpen}}$ to obtain d_h, e_h, f_h, g_h.

8: If $\tau(e_h) \neq d_h$ or $\tau(g_h) \neq f_h$ for some $h \in 1, \cdots, s$, then abort.

9: Call $\mathcal{F}_{\text{prep.rCheck}}$ on the opened values.

10: Output $(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket, \llbracket \hat{c}_i \rrbracket, \llbracket \hat{a}_i \rrbracket, \llbracket \hat{b}_i \rrbracket)$ for $i = 1, \cdots, N$.

Figure 7: Protocol for quintuple generation $\Pi_{\text{rQuintuple}}$.

Input: $\llbracket x \rrbracket, \llbracket y \rrbracket$.
Output: $\llbracket z \rrbracket$ such that $\psi(z) = \psi(x) \odot \psi(y)$.

1: Call $\mathcal{F}_{\text{prep.rQuintuple}}$ to obtain a quintuple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket, \llbracket \tau(a) \rrbracket, \llbracket \tau(b) \rrbracket)$.

2: Compute $\llbracket d \rrbracket \leftarrow \llbracket x \rrbracket - \llbracket a \rrbracket$ and $\llbracket e \rrbracket \leftarrow \llbracket y \rrbracket - \llbracket b \rrbracket$.

3: Call $\mathcal{F}_{\text{prep.rOpen}}$ to obtain $d \leftarrow \llbracket d \rrbracket$ and $e \leftarrow \llbracket e \rrbracket$.

4: Compute $\llbracket z \rrbracket \leftarrow \tau(d)\llbracket \tau(b) \rrbracket + \tau(e)\llbracket \tau(a) \rrbracket + \tau(d)\tau(e) + \llbracket c \rrbracket$.

Figure 8: Protocol for RMFE-based multiplication with a quintuple Π_{rMult}.

The functionality extends $\mathcal{F}_{\text{prep}}$, and maintains a dictionary, QVal, to keep track of the authenticated arithmetic values $[\![x]\!]^A$ for $x \in \mathbb{Z}_q$.

rDabit: On input $(\text{rDabit}, \text{id}_1, \cdots, \text{id}_k, \text{id}')$ from all parties, sample $(r_1, \cdots, r_k) \in \{0,1\}^k$ uniformly at random, and set $\text{QVal}[\text{id}_j] \leftarrow r_j$, for $j = 1, \cdots, k$, together with $\text{RVal}[\text{id}'] \leftarrow \phi([r_1, \cdots, r_k])$.

rEdabit: On input $(\text{rEdabit}, \text{id}_0, \cdots, \text{id}_{k-1}, \text{id}'_0, \cdots, \text{id}'_{\ell-1})$ from all parties ($\ell \leq \log q$), sample $(r_0, \cdots, r_{k\ell-1}) \in \mathbb{Z}_2^{k\ell}$ uniformly at random, and set $\text{RVal}[\text{id}'_j] \leftarrow \phi([r_j, r_{\ell+j}, \cdots, r_{(k-1)\ell+j}])$ for $j = 0, \cdots, \ell-1$, together with $\text{QVal}[\text{id}_i] \leftarrow \sum_{j=i\ell}^{(i+1)\ell-1} r_j 2^{j-i\ell}$ for $i = 0, \cdots, k-1$.

rEdabitPriv: On input $(\text{rEdabitPriv}, i, \text{id}_0, \cdots, \text{id}_{k-1}, \text{id}'_0, \cdots, \text{id}'_{\ell-1})$ from all parties, follow the description of rEdabit, and output $(r_0, \cdots, r_{k\ell-1})$ to party i. If party i is corrupted, it gets to choose these bits.

rB2A: On input $(\text{rB2A}, \text{id}, \text{id}'_0, \cdots, \text{id}'_{k-1})$ from all parties, set $\text{QVal}[\text{id}'_i] \leftarrow \psi(\text{RVal}[\text{id}])[\text{i}]$ for $i = 0, \cdots, k-1$.

Figure 9: Functionality $\mathcal{F}_{\text{mixed}}$

daBits remain valuable in contexts such as *boolean to arithmetic (B2A)* conversion and edaBit generation. Figure 9 outlines the minimal set of mixed-circuit functionalities used in this section.

RMFE-based daBit. In *Coral*, daBits exist in a packed version: $([\![\mathbf{r}]\!]^A, [\![r]\!])$, where $\mathbf{r} \in \{0,1\}^k$ and $r = \phi(\mathbf{r})$. We let $[\![\mathbf{r}]\!]^A$ denote a vector of authenticated arithmetic sharings in $\mathbb{Z}_q$.

RMFE-based edaBit. Packed edaBits are specified as: $([\![\mathbf{r}]\!]^A, [\![r_0]\!], \cdots, [\![r_{\ell-1}]\!])$ for an ℓ-bit RMFE-based edaBit (equivalent to k plain edaBits), where $\mathbf{r} \in \mathbb{Z}_q^k$ and $\mathbf{r}[i] = \sum_{j=0}^{\ell-1} \psi(r_j)[i] \cdot 2^j$.

Naive construction. Packed edaBits (or daBits) can be easily generated by first generating k standard edaBits (or daBits) and call $\mathcal{F}_{\text{rConv}}$ to convert the boolean part to RMFE sharings ($\mathcal{F}_{\text{rConv}}$ can be adapted to handle authenticated boolean sharings other than TinyOT sharings). Nonetheless, treating standard edaBits as an opaque intermediate step incurs a higher amortized cost.

Our construction. Escudero et al. [19] propose daBit and edaBit generation protocols applicable across various arithmetic and boolean domains. We extend these protocols (vector length k) to utilize RMFE-based MPC for the boolean component, harnessing RMFE optimizations for optimal performance. The vectorized version largely mirrors the standard one; thus, Figure 10 presents an RMFE-based edaBit generation specification, with remaining protocols detailed in Appendix D. Analogous to the original, Π_{rEdabit} constructs global packed edaBits from private ones, and correct the arithmetic part via the RMFE-based boolean-to-arithmetic functionality $\mathcal{F}_{\text{mixed.rB2A}}$ that can be implemented with packed daBits (details in the Appendix).

4.3.2 SIMD constant multiplication. In Section 2.2.4, we briefly discuss the limitation of [20] in handling linear combination. The problem stems from the case of $z \leftarrow a[\![x]\!]$ when $a \in \mathbb{F}_{2^m}$ instead of $a \in \mathbb{F}_2$. Unlike traditional constant multiplication, simple local computation here may compromise correctness: $\psi(z) \neq \psi(a) \odot \psi(x)$ because ψ is not $\mathbb{F}_{2^m}$-linear. In fact, this case turns out to be common

Output: $([\![\mathbf{r}]\!]^A, [\![r_0]\!], \cdots, [\![r_{\ell-1}]\!])$ where $\mathbf{r} \in \mathbb{Z}_q^k$ and $\mathbf{r}[i] = \sum_{j=0}^{\ell-1} \psi(r_j)[i] \cdot 2^j$.

[Construct]

1: Parties call the functionality $\mathcal{F}_{\text{mixed.rEdabitPriv}}$ to get random shares $([\![\mathbf{r}_i]\!]^A, [\![r_{i,0}]\!], \cdots, [\![r_{i,\ell-1}]\!])$ for $i = 1, \cdots, n$. Party P_i additionally learns $r_{i,j}$ and $\mathbf{r}_\mathbf{i}[h] = \sum_{j=0}^{\ell-1} \psi(r_{i,j})[h] \cdot 2^j$, for $h \in [1, k]$.

2: Parties compute $[\![\mathbf{r}']\!]^A = \sum_{i=1}^n [\![\mathbf{r}_i]\!]^A$.

3: Parties compute an ℓ-bit binary adder with n inputs $((\![r_{1,j}]\!])_j, \cdots, (\![r_{n,j}]\!])_j)$ by calling the functionality $\mathcal{F}_{\text{mpc.rMult}}$ to evaluate AND gates, obtaining $\ell + \log n$ RMFE sharings $([\![b_0]\!], \cdots, [\![b_{\ell+\log(n)-1}]\!])$.

[Correction]

4: Parties call $\mathcal{F}_{\text{mixed.rB2A}}$ to convert $[\![b_j]\!] \mapsto [\![\psi(b_j)]\!]^A$, for $j = \ell, \cdots, \ell + \log(n) - 1$. Note that $\psi(b_j)$ is a vector.

5: Parties compute $[\![\mathbf{r}]\!]^A = [\![\mathbf{r}']\!]^A - 2^\ell \sum_{j=0}^{\log(n)-1} [\![\psi(b_j)]\!]^A 2^j$.

6: Output $([\![\mathbf{r}]\!]^A, [\![b_0]\!], \cdots, [\![b_{\ell-1}]\!])$.

Figure 10: Protocol for RMFE-based edaBit generation Π_{rEdabit}.

Input: $[\![x]\!]$, $\mathbf{c}$.
Output: $[\![z]\!]$ such that $\psi(z) = \psi(x) \odot \mathbf{c}$.

1: Call $\mathcal{F}_{\text{prep.rEnc}}$ to obtain an encoding pair $([\![r]\!], [\![\tau(r)]\!])$.
2: Compute $[\![d]\!] \leftarrow [\![x]\!] - [\![r]\!]$.
3: Call $\mathcal{F}_{\text{prep.rOpen}}$ to obtain $d \leftarrow [\![d]\!]$.
4: Compute $[\![z]\!] \leftarrow (\tau(d) + [\![\tau(r)]\!]) \cdot \phi(\mathbf{c})$.

Figure 11: Protocol for RMFE-based constant multiplication with an encoding pair Π_{cMult}.

in simple circuits. For example, a general protocol paradigm for comparison is presented in many frameworks ([10, 18]): mask input $[\![x]\!]^A$ with random $[\![r]\!]^A$, open to obtain $c = x + r$, and compute a carry-out circuit between c and r's bit decomposition. Vectorizing this into a SIMD circuit would require the carry-out circuit to perform the above constant multiplication because each c is random and different inside the SIMD operation.

To fill in this gap, we employ an *encoding pair*: $([\![r]\!], [\![\tau(r)]\!])$, previously used for different purposes in previous works [9, 20]. Our SIMD constant multiplication protocol is depicted in Figure 11. Though it is more costly than non-SIMD variants due to necessitating a communication round (except when $a \in \mathbb{F}_2$), the protocol is critical for common computations. The additional expense is justified by the advantages of RMFE-based MPC and is marginal for large-scale circuit computations.

4.3.3 VOLE-style OT integration. Our framework capitalizes on recent advances in VOLE-style OT extension [6, 44] to enhance preprocessing in our protocols, a feat not fully achievable in prior research [9, 20, 27]. For example, the functionality $\mathcal{F}_{\text{COPEe}}$[3] in [27] and [9] are grounded in generalized IKNP-style OT extension [25].

[3] *Correlated oblivious product evaluation with error*, a common functionality for authenticating inputs in SPDZ-style protocols and given as Figure 15 in the Appendix.

Table 3: Complexity comparison of preprocessing.

Protocol	Comm. (Bits)	
	Input Auth.	Triple/Quintuple†
[21]	λ	$6B^2\lambda$
[20] w/ IKNP-OT	$(\lambda+1)t/k$	$12tB_1B_2^2(\lambda+1)/k$
[20] w/ VOLE-OT	$2t/k$	$24tB_1B_2^2/k$
Coral	t/k	$2(\lambda+2)B+10t/k$

†For a quintuple, this is the cost divided by RMFE packing size k

The quintuple generation in [20] relies on $\mathcal{F}_{\text{COPEe}}$ that takes two different inputs in every call, which precludes the chance of vectorizing underlying OLE calls. Instead, our preprocessing assumes a black-box access to OT extension. This allows the incorporation of VOLE-style OT extension, significantly reducing communication overhead.

TinyOT triple generation. In detail, we make use of the optimized TinyOT protocols by Wang et al. [42, 43], where only semi-honest OT extension is required to build a maliciously secure AND triple generation protocol. We re-implement the protocol by replacing the underlying OT engine with VOLE-style OT extension and integrate this into our quintuple generation protocol.

Vectorizing MFE-based offline phase. In [20], when authenticating an input $x \in \mathbb{F}_{2^m}$ from P_j with P_i holding MAC key share α, the Π_{COPEe} protocol reduces down to t calls of one-bit OLE by first applying MFE transformation to convert x and α into space $\mathbb{F}_2^t$. We instead vectorize Π_{COPEe} such that it reduces down to vector OLEs (Fig. 17), which is a natural form for the output of VOLE-style OT extension. As the vectorization is a relatively direct extension from previous works, we omit the details here and leave the vectorized functionalities, protocols and proofs in Appendix C.

4.4 Complexity Analysis

We compare *Coral*'s asymptotic complexity with [21] and [20] in the key preprocessing modules of input authentication and triple/quintuple generation. [21] (in MP-SPDZ) represents the current state-of-the-art practical solution for boolean circuits, while [20] offers a theoretical blueprint that might be instantiated with IKNP-OT or VOLE-OT configurations. Table 3 displays complexities for two parties, scaling linearly with n parties. For hyper parameters in the table, we give standard choices to have a fair comparison:

- Security parameter: $\lambda = 128$;
- (R)MFE parameters: $k = 14$, $t = 195$;
- Bucketing parameters: $B = 3$ or 4, $B_1 = B_2 = 3$ (same in [20]).

For input authentication, *Coral* demonstrates a tenfold reduction in communication complexity versus [21] and [20] with IKNP-OT, and a twofold advantage over [20] even when it employs VOLE-OT. Moreover, *Coral* exhibits a substantial, consistently order-of-magnitude improvement in communication cost for triple/quintuple generation across various hyperparameter settings, highlighting its robustness and broad applicability.

Table 4: Microbenchmarks of boolean primitives (single thread).

Protocol	Commu. (MB / 1k ops)		LAN / WAN (ms / 1k ops)	
	Input	Triple†	Input	Triple†
Tiny ([13] + [21])	0.24	66.2	3.15 / 14.80	529 / 3323
Spdz2k-SP [18]	0.24	5.08	3.15 / 14.80	122 / 385
Tinier [21]	0.016	1.84	0.17 / 2.06	10.3 / 120
*Coral** w/ [20]	0.0018	0.82	0.15 / 0.26	109 / 213
Coral	**0.0018**	**0.17**	**0.15** / **0.26**	**7.7** / **16.1**

†For *Coral** and *Coral*, this is the cost divided by RMFE packing size for 1k quintuples.

5 Evaluations

5.1 Evaluations Setup

Concrete Parameters. Our code is available at: https://github.com/AntCPLab/OpenCoral/. *Coral* is integrated into the MP-SPDZ framework [26], enhancing the framework with the following general features:

- Highly optimized mathematical support for (R)MFE;
- Packed circuit support for both the frontend compiler and backend protocols;
- Optimized TinyOT protocols for preprocessing;
- Low-communication OT from the EMP toolkit [41].

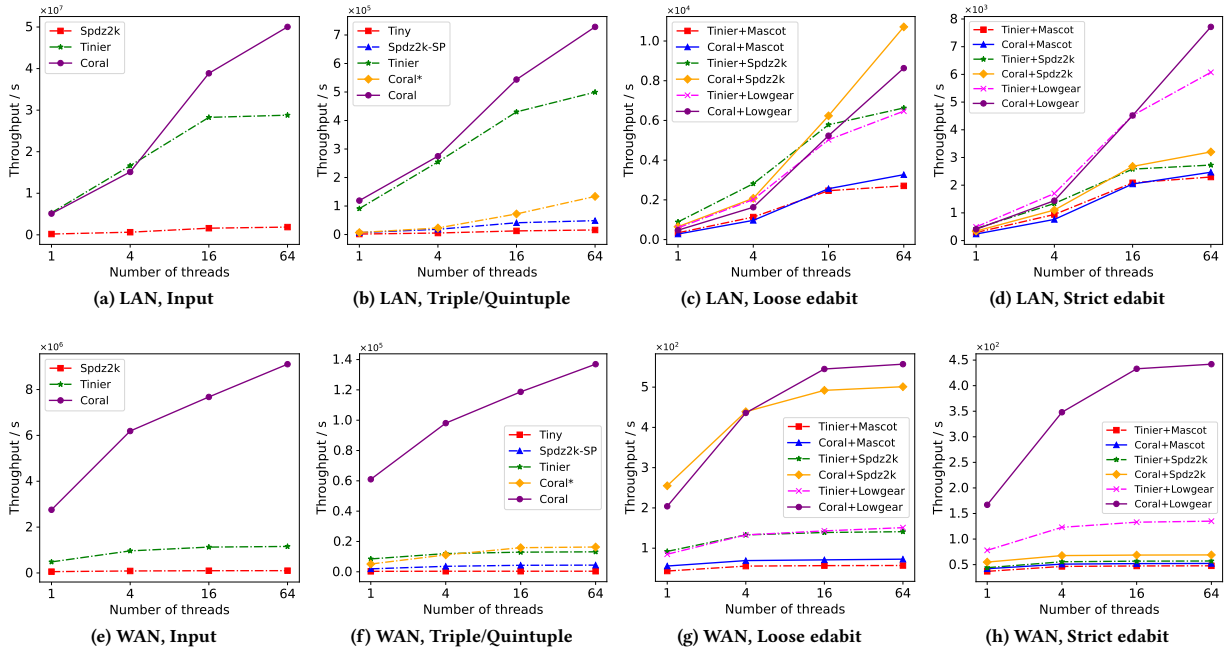
Coral can be conveniently paired with the existing arithmetic protocols in MP-SPDZ, by using our packed dabit and edabit implementation. In this section, we construct and evaluate *Coral* with minimal two parties required by the dishonest-majority model, from a compact parameter set $(14, 42)_2$-RMFE and $(195, 42)_2$-MFE, achieving an expansion ratio of $42/14 = 3$ for RMFE and $195/42 = 4.64$ for MFE-based input authentication in offline preprocessing. This parameter setting gives a statistical security level of more than 40 bits.

Testbed Environment. The experiments are conducted on commercial cloud instances featuring 2.7GHz processors and 64GB RAM, using Linux's traffic control command to adjust bandwidth and latency. Benchmarks run under two network configurations: a LAN with 10Gbps and 0.2ms latency, and a WAN with 200Mbps and 20ms latency.

Metrics. We measure end-to-end running time including both computation and network IO. We measure the total communication including all the messages sent by the parties.

5.2 Microbenchmarks

5.2.1 Boolean primitives. Maliciously secure boolean protocols consist of two expensive submodules: input authentication and multiplication randomness generation (including triples and quintuples). Since there exist several possible combinations, we choose four baselines for a complete comparison: Tinier and Tiny are available in MP-SPDZ library. The former is implemented based on [21]. The latter authenticates inputs by using Spdz2k [13], but uses the triple generation technique from [21]. We implement Spdz2k-SP based on [18] that authenticates inputs using [13], but generates triples by converting from TinyOT triples. We also implement *Coral**



(a) LAN, Input (b) LAN, Triple/Quintuple (c) LAN, Loose edabit (d) LAN, Strict edabit

(e) WAN, Input (f) WAN, Triple/Quintuple (g) WAN, Loose edabit (h) WAN, Strict edabit

Figure 12: Preprocessing material generation throughput across different protocols, networks and number of threads.

which reuses all of *Coral*'s settings but generates quintuples according to [20]. The reported costs are amortized for 1000 operations.

Initial tests on microbenchmarks using a single thread show that *Coral* significantly outperforms the previously most efficient protocol, Tinier, with communication overhead improvements of approximately 10×. Specifically, it achieves an 89% reduction in input authentication (from 0.016 MB to 0.0018 MB) and a 91% reduction in triple generation (from 1.84 MB to 0.17 MB). This improvement notably enhances runtime performance in actual network conditions. For instance, in a WAN setting, *Coral* is over 6× faster than Tinier. However, in LAN settings, the gains are somewhat mitigated by the added computational demands of (R)MFE encoding and decoding. Additionally, our quintuple generation protocol exhibits efficiency over an order of magnitude higher than that of [20]. When compared to Spdz2k-SP, which also uses TinyOT triples, *Coral* proves to be significantly more efficient largely due to our streamlined, vectorized MFE-based input authentication that complements TinyOT conversion effectively.

Figure 12 illustrates *Coral*'s preprocessing throughput in multithreaded settings, from 1 to 64 threads, highlighting its practical efficiency. In LAN, *Coral* consistently outperforms or matches Tinier, especially at higher levels of parallelism. In WAN, *Coral*'s throughput is 5-8× and up to 10× greater than Tinier for input authentication and triple generation, respectively. Notably, the less efficient variant, *Coral**, surpasses Tinier with 16 or more threads, benefiting from its reduced reliance on network bandwidth, which hampers more communication-heavy protocols.

5.2.2 Mixed-circuit primitives. Beyond purely boolean computation, we explore how *Coral* enhances mixed-circuit computing with packed dabit and edabit. We compare the two leading boolean protocols, Tinier and *Coral*, paired with top arithmetic protocols from MP-SPDZ: Mascot [27], Lowgear [28] and Spdz2k [13]. Mascot and Lowgear operate within prime fields, while Spdz2k works in ring $\mathbb{Z}_{2^k}$. For prime fields, we set $\log p \approx 128$ to allow additional room needed for comparison [19], and use $k = 64$ for mod 2^k (extended to $k + s = 128$ for malicious security). The preprocessing for Mascot and Spdz2k uses oblivious transfer, while Lowgear relies on homomorphic encryption. This assortment allows a comprehensive assessment of *Coral* across various mixed-circuit integrations.

Table 5: Microbenchmarks of edabits (single thread).

Protocol	Commu. (MB/1k ops)		LAN / WAN (s/1k ops)	
	loose	strict	loose	strict
Tinier+[27] (p, OT)	436	522	2.82 / 23	3.32 / 26.9
Coral+[27] (p, OT)	348	479	3.28 / 17.9	4.07 / 23.8
Tinier+[13] (2^k, OT)	175	435	1.12 / 10.8	2.23 / 22.5
Coral+[13] (2^k, OT)	50.1	362	1.45 / 3.88	2.79 / 18.1
Tinier+[28] (p, HE)	169	181	1.69 / 11.7	2.02 / 12.8
Coral+[28] (p, HE)	42.1	53.6	1.89 / 4.6	2.23 / 5.6

Dabit generation exhibited only marginal gains with our approach, primarily due to the cost of arithmetic multiplication triples (used in generating arithmetic random bits [18] or checking dabits [19]), so we focus on the more critical results for edabits crucial for mixed-circuit computations. Following previous work [19], we distinguish *loose* and *strict* edabits, with the latter involving full generation and contributing to the major costs of fixed-point truncation or

Table 6: Evaluated boolean circuits. I_1, I_2 and O are bit length of two inputs and the output. The total number of gates and the number of AND gates are also listed.

Circuit	I_1	I_2	O	Total	AND
Hamming Dist.	1024	1024	12	8148	2036
AES	128	128	128	36663	6400
SHA256	512	256	256	135073	22573
Sorting	4096	4096	4096	643681	114688

comparison in prime field. Loose edabits, suitable for $\mathbb{Z}_{2^\ell}$ comparisons, skip the correction step. Table 5 presents benchmarks for mixed-circuit schemes in generating both types. Notably, *Coral*'s performance gain varies with arithmetic scheme pairing, excelling with Lowgear (HE-based), offering 73.6% communication reduction and 2.5× speedup in single-threaded WAN for loose edabits. HE-based protocols generally exhibit lower communication overhead than OT-based ones, amplifying *Coral*'s impact on boolean computation. Improvement in strict edabit generation is less pronounced due to heavier reliance on multiplication triples. In LAN, *Coral* underperforms with a single thread but multithreading mitigates this disadvantage, as demonstrated subsequently.

Figure 12 illustrates edabit generation throughput in multithreaded environments. In LAN, *Coral*-supported protocols increasingly surpass Tinier-based counterparts with rising thread count. This performance gap expands in WAN, reaching nearly 4× higher throughput with *Coral*-based protocols at 16 threads, evidencing *Coral*'s superior concrete efficiency for large circuit evaluation.

5.3 Boolean Circuits

The results of the previous section show that our protocol is more efficient in various primitives. In this section, we demonstrate *Coral*'s capability in boosting the performance of concrete boolean circuit evaluation. We evaluate Tinier, *Coral** and *Coral* on the circuits that have been used in previous works [42]: 1) **Hamming distance** between two n-bit strings using an $O(n)$-size circuit where $n = 1024$, 2) **AES-128** circuit, 3) **SHA256** circuit, 4) **Bitonic sorting** on an array of 128 elements of 32 bits. Table 6 lists the details of these circuits, either provided by SCALE-MAMBA [30] or generated from EMP toolkit [41]. Our experiments are organized in the *amortized* setting: the reported results are amortized for a single instance over a packed execution of 140 instances so that little waste of buffered processing material is observed.

In Table 7 we show the performance on the above examples. Compared to Tinier, the communication improvement is consistent with the results of Table 5, giving over one order of magnitude improvement factor. For hamming distance that has a very small circuit size, we observe a higher waste of preprocessing material in *Coral* (due to the packed nature of our method), which results in a lower factor of 10×. The factor goes up to 11.4× for the large sorting circuit, which is explained by a full use of preprocessing material and a low amortized cost of EMP's `Ferret` OT (*Coral*'s results in Table 4 waste a certain amount of batch-generated material in `Ferret`). Compared to *Coral**, *Coral* still achieves 5× improvement in communication. When the network is fast, *Coral* has similar runtime performance as Tinier. But under the WAN setting, *Coral* improves over Tinier by a factor of 4.4×-7.2×, ranging from small circuits to large circuits that we evaluate. On the contrary, in spite of having lower communication than Tinier, *Coral** has poor performance due to its complicated quintuple generation protocol. *Coral* achieves up to 30× speedup over *Coral** in end-to-end circuit evaluation, demonstrating the high concrete efficiency of our protocols.

5.4 Mixed Circuits

The MP-SPDZ library provides a convenient compiler to translate machine learning applications written in a python-like language into low-level multiparty computation instructions. To fully support *Coral* in an optimal way, we update the ML compiler to enable packing and evaluate multiple inputs simultaneously, which ensures that *Coral*-based protocols are used with minimal waste of computation and communication. In Table 8 we run applications of various scales with a *Coral*-based protocol and a Tinier-based protocol. Based on results in Section 5.2.2, we choose the two best-performing protocols paired with Lowgear. We list some details of the applications below:

- The decision tree model is relatively small with depth of 6 and predicts samples from the breast cancer dataset with 30 features.
- Lenet Small and Lenet Large have the same architecture of 6 neural network layers, with different number of internal neurons. They evaluate standard MNIST images of dimension 28×28.
- SqueezeNet has 22 layers and evaluate ImageNet images of dimension $227 \times 227 \times 3$.
- ResNet has 50 layers and evaluate ImageNet images of dimension $224 \times 224 \times 3$ (specifically, we use ResNet-50 v2 with FP32 precision).

Applications are executed with parallelism levels tailored to their sizes, and results are normalized per single input evaluation. *Coral* achieves over 50% reduction in communication overhead for entire applications through non-linear computation optimization. This significantly enhances performance in the typical WAN setting for malicious security, with an average 3× speedup across all applications. Surprisingly, LAN runtime improvements exceed expectations for certain ML applications, partly due to our ML compiler's packing optimizations.

SqueezeNet and ResNet suffer severe bottlenecks in WAN conditions. We estimate a lower bound on runtime based on bandwidth and provide comprehensive insights into non-linear computation that involves *Coral* and Tinier by using actual throughput of boolean triples, dabits, and edabits under 64 threads. The compiler-generated preprocessing materials for these networks' non-linear parts are detailed in Appendix E. Analogous results reveal more pronounced communication overhead reductions for pure non-linear components. Despite these improvements, arithmetic computation remains the dominant cost in numerous ML applications, necessitating complementary research efforts, as exemplified by recent advancements in matrix and convolution triples [11, 36]. Integrating such enhancements and elevating the performance of maliciously secure mixed-circuit computation represents a promising avenue for future work.

Table 7: Experimental results for boolean circuits

Circuit	Commu. (MB)			LAN (s)			WAN (s)		
	Tinier	*Coral**	*Coral*	Tinier	*Coral**	*Coral*	Tinier	*Coral**	*Coral*
Hamming Dist.	3.81	1.69	0.38	0.017	0.25	0.016	0.22	0.88	0.05
AES	11.9	5.14	1.07	0.054	0.76	0.046	0.67	2.69	0.11
SHA256	41.6	18.1	3.65	0.187	2.72	0.138	2.56	9.57	0.45
Sorting	210.8	91.9	18.5	0.927	14.7	0.65	11.8	48.1	1.65

Table 8: Performance comparison of *Coral* and Tinier paired with Lowgear in mixed circuits.

Application	Commu.			LAN			WAN		
	Tinier-LG	*Coral*-LG	Factor	Tinier-LG	*Coral*-LG	Factor	Tinier-LG	*Coral*-LG	Factor
DTree (1 thread)	19.7 MB	8.87 MB	2.2×	0.36 s	0.32 s	1.1×	5.92 s	1.2 s	4.9×
Lenet Small (1 thread)	13.7 GB	4.55 GB	3×	185 s	159 s	1.2×	1689 s	453 s	3.7×
Lenet Large (8 threads)	90.2 GB	39.8 GB	2.3×	374 s	182 s	2.1×	5038 s	1740 s	2.9×
SqueezeNet (64 threads)	4 TB	1.69 TB	2.4×	2.3 h	0.78 h	2.9×	> 46 h†	> 19 h†	-
SqueezeNet NonLinear Prep. (64 threads)‡	2.52 TB	0.87 TB	2.9×	0.69 h	0.59 h	1.2×	28.5 h	10 h	2.9×
ResNet (64 threads)	21.5 TB	13.8 TB	1.6×	10 h	5.1 h	2×	> 248 h†	> 158 h†	-
ResNet NonLinear Prep. (64 threads)‡	8.86 TB	3.2 TB	2.8×	2.4 h	2 h	1.2×	98.3 h	36.3 h	2.7×

†Estimated based on LAN running time and network transmission under WAN's bandwidth 200Mbps.
‡Extrapolated based on preprocessing material requirement and throughput under 64 threads (Appendix E).

6 Conclusion

We study and develop RMFE-based MPC in detail, filling in all the missing pieces in previous works. The resulting implementation for MFE and RMFE is useful for evaluation of future works in this research line. The proposed new framework *Coral* boosts the concrete efficiency of existing boolean circuits and mixed circuits by a great scale. We also see that for large applications, more research and more engineering effort are expected for practical efficiency in the considered strong security setting.

Acknowledgments

We thank Chaoping Xing and Chen Yuan for their valuable comments on this work.

References

[1] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. 2019. Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE. In *WAHC@CCS*. ACM, 33–44.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *CCS*. New York, NY, USA, 535–548.

[3] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*, Vol. 6632. 169–188.

[4] Alexander R. Block, Hemanta K. Maji, and Hai H. Nguyen. 2018. Secure Computation with Constant Communication Overhead Using Multiplication Embeddings. In *INDOCRYPT*, Vol. 11356. 375–398.

[5] Charlotte Bonte, Nigel P. Smart, and Titouan Tanguy. 2021. Thresholdizing HashEdDSA: MPC to the Rescue. *Int. J. Inf. Sec.* 20, 6 (2021), 879–894.

[6] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In *CCS*. ACM, 291–308.

[7] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. 2021. High-Performance Multi-party Computation for Binary Circuits Based on Oblivious Transfer. *J. Cryptol.* 34, 3 (2021), 34.

[8] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. 2018. Amortized Complexity of Information-Theoretically Secure MPC Revisited. In *CRYPTO (3) (Lecture Notes in Computer Science, Vol. 10993)*. Springer, 395–426.

[9] Ignacio Cascudo and Jaron Skovsted Gundersen. 2020. A Secret-Sharing Based MPC Protocol for Boolean Circuits with Good Amortized Complexity. In *TCC (2) (Lecture Notes in Computer Science, Vol. 12551)*. Springer, 652–682.

[10] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN*, Vol. 6280. Springer, 182–199.

[11] Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *ASIACRYPT 2020*. 31–59.

[12] Jung Hee Cheon, Dongwoo Kim, and Keewoo Lee. 2021. MHz2k: MPC from HE over $\mathbb{Z}_{2^k}$ with New Packing, Simpler Reshare, and Better ZKP. In *CRYPTO 2021 (Lecture Notes in Computer Science)*. 426–456.

[13] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO (2) (Lecture Notes in Computer Science, Vol. 10992)*. Springer, 769–798.

[14] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. 2021. Asymptotically-Good Arithmetic Secret Sharing over $\mathbb{Z}/p^\ell\mathbb{Z}$ with Strong Multiplication and Its Applications to Efficient MPC. In *CRYPTO (3) (Lecture Notes in Computer Science, Vol. 12827)*. Springer, 656–686.

[15] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*, Vol. 8134. 1–18.

[16] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*. 643–662.

[17] Ivan Damgård and Sarah Zakarias. 2013. Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing. In *TCC*, Vol. 7785. 621–641.

[18] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1102–1120.

[19] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO*. 823–852.

[20] Daniel Escudero, Chaoping Xing, and Chen Yuan. 2022. More Efficient Dishonest Majority Secure Computation over $\mathbb{Z}^{2^k}$ via Galois Rings. In *CRYPTO*. 383–412.

[21] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. A Unified Approach to MPC with Preprocessing Using OT. In *ASIACRYPT (1) (Lecture Notes in Computer Science, Vol. 9452)*. Springer, 711–735.

The functionality maintains a dictionary, TVal, to keep track of the authenticated values $[\![\cdot]\!]^B$ in TinyOT sharing, and RVal, to keep track of the authenticated values $[\![\cdot]\!]$ in RMFE sharing.

tOpen: On input $(\mathsf{tOpen}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{TVal.keys}()$, send $\mathsf{TVal}[\mathsf{id}]$ to the adversary and wait for x from the adversary, and output x to all parties.

tCheck: On input $(\mathsf{tCheck}, \mathsf{id}_1, \cdots, \mathsf{id}_t, x_1, \cdots, x_t)$ from all parties, wait for an input from the adversary. If it inputs OK, and $\mathsf{TVal}[\mathsf{id}_j] = x_j$ for all j, return OK to all parties, otherwise abort.

rInput: On input $(\mathsf{rInput}, \mathsf{id}_1, \cdots, \mathsf{id}_l, x_1, \cdots, x_l, P_j)$ from party P_j and $(\mathsf{Input}, \mathsf{id}_1, \cdots, \mathsf{id}_l, P_j)$ from all other parties, where $x_i \in \mathbb{F}_{2^m}$, set $\mathsf{RVal}[\mathsf{id}_i] \leftarrow x_i$ for $i = 1, \cdots, l$.

rOpen: On input $(\mathsf{rOpen}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{RVal.keys}()$, send $\mathsf{RVal}[\mathsf{id}]$ to the adversary and wait for x from the adversary, and output x to all parties.

rCheck: On input $(\mathsf{rCheck}, \mathsf{id}_1, \cdots, \mathsf{id}_t, x_1, \cdots, x_t)$ from all parties, wait for an input from the adversary. If it inputs OK, and $\mathsf{RVal}[\mathsf{id}_j] = x_j$ for all j, return OK to all parties, otherwise abort.

Figure 13: Functionality $\mathcal{F}_{\text{prep}}$

The functionality maintains a dictionary, Val, to keep track of the authenticated values of $\mathbb{F}_2^k$.

rAffComb: On input $(\mathsf{rAffComb}, \mathsf{id}, (\mathsf{id}_1, \cdots, \mathsf{id}_L), (a_1, \cdots, a_L), \mathbf{a})$ from all parties, where $a_j \in \mathbb{F}_2$ for $j = 1, \cdots, L$ and $\mathbf{a} \in \mathbb{F}_2^k$, computes and sets $\mathsf{Val}[\mathsf{id}] \leftarrow \mathbf{a} + \sum_{j=1}^{L} a_j \cdot \mathsf{Val}[\mathsf{id}_j]$.

cMult: On input $(\mathsf{cMult}, \mathsf{id}, \mathsf{id}', \mathbf{c})$, where $\mathbf{c} \in \mathbb{F}_2^k$, computes and sets $\mathsf{Val}[\mathsf{id}] \leftarrow \mathbf{c} \odot \mathsf{Val}[\mathsf{id}']$.

rMult: On input $(\mathsf{rMult}, \mathsf{id}, (\mathsf{id}_1, \mathsf{id}_2))$ from all parties, computes and sets $\mathsf{Val}[\mathsf{id}] \leftarrow \mathsf{Val}[\mathsf{id}_1] \odot \mathsf{Val}[\mathsf{id}_2]$.

Figure 14: Functionality $\mathcal{F}_{\text{mpc}}$

[22] Tore Kasper Frederiksen, Benny Pinkas, and Avishay Yanai. 2018. Committed MPC - Maliciously Secure Multiparty Computation from Homomorphic Commitments. In *Public Key Cryptography (1)*, Vol. 10769. 587–619.

[23] Tom Albæk Hansen and Gary L. Mullen. 1992. Primitive polynomials over finite fields. *Math. Comp.* 59 (1992), 639–643.

[24] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO (Lecture Notes in Computer Science, Vol. 2729)*. Springer, 145–161.

[25] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO*. 145–161.

[26] 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*.

[27] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *CCS*. ACM, 830–842.

[28] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *EUROCRYPT 2018*. 158–189.

[29] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. 2014. Dishonest Majority Multi-Party Computation for Binary Circuits. In *CRYPTO (2)*, Vol. 8617. 495–512.

[30] KU Leuven. 2021. SCALE and MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA/.

[31] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 681–700.

[32] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. 2020. Overdrive2k: Efficient Secure MPC over $\mathbb{Z}_{2^k}$ from Somewhat Homomorphic Encryption. In *CT-RSA*, Vol. 12006. Springer, 254–283.

[33] Ignacio Cascudo Pueyo, Hao Chen, Ronald Cramer, and Chaoping Xing. 2009. Asymptotically Good Ideal Linear Secret Sharing with Strong Multiplication over *Any* Fixed Finite Field. In *CRYPTO (Lecture Notes in Computer Science, Vol. 5677)*. Springer, 466–486.

[34] Microsoft Research. 2024. EzPC: Easy Secure Multiparty Computation. https://github.com/mpc-msri/EzPC.

[35] Peter Rindal. 2024. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[36] Marc Rivinius, Pascal Reisert, Sebastian Hasler, and Ralf Küsters. 2023. Convolutions in Overdrive: Maliciously Secure Convolutions for MPC. *Proc. Priv. Enhancing Technol.* 2023, 3 (2023), 321–353.

[37] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. 2022. Actively Secure Setup for SPDZ. *J. Cryptol.* 35, 1 (2022), 5.

[38] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *INDOCRYPT*, Vol. 11898. Springer, 227–249.

[39] Victor Shoup. 2023. NTL: A Library for doing Number Theory. https://libntl.org/.

[40] Berk Sunar, Erkay Savas, and Çetin Kaya Koç. 2003. Constructing Composite Field Representations for Efficient Conversion. *IEEE Trans. Computers* 52, 11 (2003), 1391–1398.

[41] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2022. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[42] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *CCS*. ACM, 21–37.

[43] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. In *CCS*. ACM, 39–56.

[44] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast Extension for Correlated OT with Small Communication. In *CCS*. 1607–1626.

A Missing functionalities

We detail the standard functionalities that are missing in the main text. Figure 13 list the missing commands in $\mathcal{F}_{\text{prep}}$. Figure 14 list the missing functionality for the online phase of MPC $\mathcal{F}_{\text{mpc}}$. It is partially taken from [20], with an additional command cMult which describes the SIMD constant multiplication missing in the previous work and implemented by our protocol Π_{cMult}.

B Missing proofs

THEOREM 6. *Π_{rConv} securely implements $\mathcal{F}_{\text{rConv}}$ in the $\mathcal{F}_{\text{prep}}$-hybrid (Figure 4) model with statistical security parameter s.*

PROOF. The proof is similar to the argument in [18, 20]. $\mathcal{F}_{\text{rConv}}$ requires the conversion output to be both consistent and normal (note that $\tau(\phi(\mathbf{x})) = \phi(\mathbf{x})$). Suppose the adversary causes incorrect values y'_j or q'_j to be authenticated as RMFE sharings. There are two possible types of incorrectness:

- $\tau(y'_j) \neq y'_j$, i.e., y'_j is *not* a normal element and thus $y'_j \neq y_j$.
- $\psi(y'_j) \neq \psi(y_j)$, *i.e.*, $\psi(y'_j)[h] \neq x_{(j-1)k+h}$ for some $h \in [1, k]$;

Check normality. Let $\tau(y'_j) = y'_j + \beta_j$, $\tau(q'_j) = q'_j + \gamma_j$. Assume that there exists some $\beta_j \neq 0$, then the equality $\sum_{j=1}^{n} c_{j,g} \cdot \beta_j + \gamma_g = 0$ holds with probability at most 1/2 for a certain g because each $c_{j,g}$ is uniformly random and independent of β_j and γ_g. The adversary succeeds with probability as most 2^{-s} by passing all s checks.

Check consistency. Let $\psi(y'_j)[h] = x_{(j-1)k+h} + \delta_{j,h} \mod 2$, $\psi(q'_j)[h] = r_{(j-1)k+h} + \epsilon_{j,h} \mod 2$. Assume that there exists some $\delta_{j,h} \neq 0$, then the equality $\sum_{j=1}^{n} c_{j,g} \cdot \delta_{j,h} + \epsilon_{g,h} = 0$ holds with probability at most 1/2 for a certain (g, h) pair because each $b_{j,g}$ is uniformly random and independent of $\delta_{j,h}$ and $\epsilon_{g,h}$. Note that $\psi(w_g) = \sum_{j=1}^{n} c_{j,g} \psi(y'_j) + \psi(q_g)$. The adversary succeeds with probability as most 2^{-s} by passing all s checks.

Taking both checks into account, it follows that the outputs are correct with probability at least $1 - 2^{-s}$. □

The functionality runs with two parties P_i and P_j and the adversary $\mathcal{A}$. The **Initialize** phase is run once first. The **Multiply** phase can be run an arbitrary number of times.
Initialize: On input $\alpha^{(i)} \in \mathbb{F}_{2^m}$ from P_i, store this value.
Multiply: On input $x \in \mathbb{F}_{2^m}$ from P_j:

- If P_j is corrupt then receive $v^{(j,i)} \in \mathbb{F}_{2^m}$ and a vector $\mathbf{x}^{(j,i)} \in \mathbb{F}_2^t$ from $\mathcal{A}$ and compute $u^{(i,j)} = \rho(\sigma(\alpha^{(i)}) \odot \mathbf{x}^{(j,i)}) - v^{(j,i)}$.
- If P_i is corrupt then receive $\boldsymbol{\alpha}^{(i,j)} \in \mathbb{F}_2^t$ and $u^{(i,j)}$ from $\mathcal{A}$ and compute $v^{(j,i)} = \rho(\boldsymbol{\alpha}^{(i,j)} \odot \sigma(x)) - u^{(i,j)}$
- If both P_i and P_j are honest then sample $u^{(i,j)}$ and $v^{(j,i)}$ uniformly at random subject to $u^{(i,j)} + v^{(j,i)} = \alpha^{(i)} \cdot x$.

The functionality sends $u^{(i,j)}$ to P_i and $v^{(j,i)}$ to P_j.

Figure 15: Functionality $\mathcal{F}_{\text{COPEe}}$ [20]

The functionality runs with two parties P_i and P_j and the adversary $\mathcal{A}$. The **Initialize** phase is run once first. The **Multiply** phase can be run an arbitrary number of times.
Initialize: On input $\alpha^{(i)} \in \mathbb{F}_{2^m}$ from P_i, store this value.
Multiply: On input $\mathbf{x} \in \mathbb{F}_{2^m}^L$ from P_j:

- If P_j is corrupt then receive $\mathbf{v}^{(j,i)} \in \mathbb{F}_{2^m}^L$ and a matrix $\mathbf{X}^{(j,i)} \in \mathbb{F}_2^{L\times t}$ from $\mathcal{A}$ and compute $\mathbf{u}^{(i,j)} = \rho(\sigma(\alpha^{(i)}) \odot \mathbf{X}^{(j,i)}) - \mathbf{v}^{(j,i)}$.
- If P_i is corrupt then receive $\boldsymbol{\alpha}^{(i,j)} \in \mathbb{F}_2^t$ and $\mathbf{u}^{(i,j)} \in \mathbb{F}_{2^m}^L$ from $\mathcal{A}$ and compute $\mathbf{v}^{(j,i)} = \rho(\boldsymbol{\alpha}^{(i,j)} \odot \sigma(\mathbf{x})) - \mathbf{u}^{(i,j)}$
- If both P_i and P_j are honest then sample $\mathbf{u}^{(i,j)}$ and $\mathbf{v}^{(j,i)}$ uniformly at random subject to $\mathbf{u}^{(i,j)} + \mathbf{v}^{(j,i)} = \alpha^{(i)} \cdot \mathbf{x}$.

The functionality sends $\mathbf{u}^{(i,j)}$ to P_i and $\mathbf{v}^{(j,i)}$ to P_j.

Figure 16: Functionality $\mathcal{F}_{\text{vCOPEe}}$

Upon receiving $(\text{vOLE}, a, P_A, P_B)$ from P_A and $(\text{vOLE}, \mathbf{x}, P_A, P_B)$ from P_B where $a \in \mathbb{F}_2$, $\mathbf{x} \in \mathbb{F}_2^L$, the functionality samples uniformly random $\mathbf{b} \leftarrow \mathbb{F}_2^L$, sets $\mathbf{y} = a\mathbf{x} + \mathbf{b}$, and sends $\mathbf{y}$ to P_B and $-\mathbf{b}$ to P_A.

Figure 17: Functionality $\mathcal{F}_{\text{vOLE}}$

Initialize: On input $\alpha^{(i)} \in \mathbb{F}_{2^m}$ from P_i
Multiply: On input $\mathbf{x} \in \mathbb{F}_{2^m}^L$ from P_j:

1: P_i computes $\sigma(\alpha^{(i)}) = (a_1, \cdots, a_t) \in \mathbb{F}_2^t$, and P_j computes $\sigma(\mathbf{x}) = \mathbf{X} \in \mathbb{F}_2^{L\times t}$. P_i defines an empty matrix $\mathbf{B} \in \mathbb{F}_2^{L\times t}$ and P_j defines an empty matrix $\mathbf{Y} \in \mathbb{F}_2^{L\times t}$.
2: For $h = 1, \cdots, t$, P_i and P_j call $\mathcal{F}_{\text{vOLE}}$ where P_i inputs a_h and P_j inputs a column $\mathbf{X}_{*h}$. P_j receives $\mathbf{Y}_{*h}$, and P_i receives $-\mathbf{B}_{*h} = a_h \cdot \mathbf{X}_{*h} - \mathbf{Y}_{*h}$.
3: P_i sets $\mathbf{u}^{(i,j)} = -\rho(\mathbf{B})$ and P_j sets $\mathbf{v}^{(i,j)} = \rho(\mathbf{Y})$.

Figure 18: Protocol Π_{vCOPEe}.

Theorem 7. *$\Pi_{\text{rQuintuple}}$ securely implements $\mathcal{F}_{\text{prep.rQuintuple}}$ in the $(\mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{rConv}})$-hybrid model with s-bit statistical security.*

Proof. The proof is similar to the argument in [20]. $\mathcal{F}_{\text{rConv}}$ produces consistent and normal triples $(\llbracket \hat{a_j} \rrbracket, \llbracket \hat{b_j} \rrbracket, \llbracket \hat{c_j} \rrbracket)$. The remaining part of the protocol is to generate random a_j and b_j such that $\tau(a_j) = \hat{a_j}$ and $\tau(b_j) = \hat{b_j}$. Suppose the adversary causes incorrect values a'_j or b'_j to be authenticated as RMFE sharings.

Let $\tau(a'_j) = \hat{a_j} + \delta_j$. Assume that there exists some $\delta_j \neq 0$ for $j \in [1, N]$, then the equality $\sum_{j=1}^{N} r_{h,j} \cdot \delta_j + \delta_{N+h} = 0$ holds with probability at most 1/2 for a certain h because each $r_{h,j}$ is uniformly random and independent of δ_j and δ_{N+h}. The adversary succeeds with probability as most 2^{-s} by passing all s checks. Similarly, the same analysis applies for b'_j. □

Theorem 8. *Π_{rMult} securely implements $\mathcal{F}_{\text{mpc.rMult}}$ in the $\mathcal{F}_{\text{prep}}$-hybrid model.*

Proof. All steps in the protocol resemble those from [20], except step 4 which is pure local computation, hence we only show its correctness here and refer the readers to [20] for the complete security proof. We see that $z = \tau(d)\tau(b) + \tau(e)\tau(a) + \tau(d)\tau(e) + c = \tau(x)\tau(y) - \tau(a)\tau(b) + c$. Therefore, we have:

$$\begin{aligned}\psi(z) &= \psi(\tau(x)\tau(y)) - \psi(\tau(a)\tau(b)) + \psi(c)\\ &= \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y})) - \psi(\phi(\mathbf{a}) \cdot \phi(\mathbf{b})) + \psi(a) \odot \psi(b)\\ &= \mathbf{x} \odot \mathbf{y} - \mathbf{a} \odot \mathbf{b} + \mathbf{a} \odot \mathbf{b} = \mathbf{x} \odot \mathbf{y}\end{aligned}$$

□

C Vectorized COPEe

In [20], the COPEe functionality is defined with MFE encoding σ and decoding ρ embedded (Figure 15). In fact, this functionality can be vectorized to represent multiple independent calls, which we shown in Figure 16. Such a vectorized usage is present in the input authentication protocol (Π_{Auth} in [20]) that can be updated to use our vectorized functionality, and the proof can be reused. The $\mathcal{F}_{\text{vCOPEe}}$ functionality can be implemented with an access to $\mathcal{F}_{\text{vOLE}}$ (Figure 17). The protocol Π_{vCOPEe} (Figure 18) is a vectorized version of Π_{COPEe} in [20]. We give the adapted proof below.

Theorem 9. *Π_{vCOPEe} securely implements $\mathcal{F}_{\text{vCOPEe}}$ in the $\mathcal{F}_{\text{vOLE}}$-hybrid model.*

Proof. If P_i is corrupted, the simulator receives $\boldsymbol{\alpha}^{(i,j)} \in \mathbb{F}_2^t$ from the adversary. The simulator samples $\mathbf{B} \leftarrow \mathbb{F}_2^{L\times t}$ uniformly at random and sends $\mathbf{B}$ to the adversary.

If P_j is corrupted, the simulator receives $\mathbf{X} \in \mathbb{F}_2^{L\times t}$ from the adversary. The simulator samples $\mathbf{Y} \leftarrow \mathbb{F}_2^{L\times t}$ uniformly at random and sends $\mathbf{Y}$ to the adversary. The indistinguishability is clear since the output looks uniformly at random both in the real world and the ideal world. □

The importance of such vectorization is that $\mathcal{F}_{\text{vOLE}}$ can be implemented efficiently with correlated OT based on recent advances in VOLE-style OT extension [6, 44]. Note that the previous input authentication protocol [20] could also potentially benefit from this, but their quintuple generation fails to enjoy this benefit due to non-vectorizable usage.

Input: RMFE sharings $[\![x]\!]$, where $\psi(x) = \mathbf{x} = [x_0, x_1, \cdots, x_{k-1}]$
Output: Arithmetic sharings $[\![\mathbf{x}]\!]^A = [\![x_0]\!]^A, [\![x_1]\!]^A, \cdots, [\![x_{k-1}]\!]^A$

1: Call $\mathcal{F}_{\text{mixed.rDabit}}$ to obtain a packed daBit: $[\![\mathbf{r}]\!]^A, [\![r]\!]$.
2: Call $\mathcal{F}_{\text{prep.rOpen}}$ on $[\![x]\!] + [\![r]\!]$ and obtain $c_0, \cdots, c_{k-1} = \psi(x + r)$.
3: Compute $[\![x_i]\!]^A = c_i + [\![r_i]\!]^A - 2 \cdot c_i \cdot [\![r_i]\!]^A$.

Figure 20: Protocol for RMFE-based boolean to arithemetic Π_{rB2A}.

Output: N packed edaBits $\{([\![\mathbf{r_j}]\!]^A, [\![r_{j,0}]\!], \cdots, [\![r_{j,\ell-1}]\!])\}_{j=1}^N$, where $\mathbf{r_j} \in \mathbb{Z}_q^k$ and $\mathbf{r_j}[i] = \sum_{g=0}^{\ell-1} \psi(r_{j,g})[i] \cdot 2^g$, and P_i knows the underlying bits.
[Construct]

1: P_i samples $b_{j,0}, \cdots, b_{j,k\ell-1} \in \mathbb{Z}_2$, and calls $\mathcal{F}_{\text{prep.rInput}}$ to obtain $[\![r_{j,i}]\!]$, where $r_{j,i} = \phi([b_{j,i}, b_{j,\ell+i}, \cdots, b_{j,(k-1)\ell+i}])$, for $i = 0, \cdots, \ell-1$ and $j = 1, \cdots, NB + C$.
2: P_i computes $\mathbf{r_j}[i] = \sum_{g=0}^{\ell-1} b_{j,i\ell+g} 2^g$ and call $\mathcal{F}_{\text{prep.qInput}}$ to obtain $[\![\mathbf{r_j}[i]]\!]^A$.
3: P_i samples $(N(B-1) + C')\ell$ random quintuples and call $\mathcal{F}_{\text{prep.rInput}}$ to inputs these.

[Cut and Choose]

4: Parties sample two public random permutations and use these to shuffle the packed edaBits and the private quintuples.
5: Open the first C of the shuffled packed edaBits, and the first $C' \cdot \ell$ quintuples. Abort if any of the packed edaBits or the quintuples are inconsistent.
6: Place the remaining packed edaBits into buckets of size B and the quintuples into bucket of size $(B-1) \cdot \ell$.
7: For each bucket, select the first packed edaBit $([\![\mathbf{r}]\!]^A, [\![r_0]\!], \cdots, [\![r_{\ell-1}]\!])$, and for the every other packed edaBit $([\![\mathbf{s}]\!]^A, [\![s_0]\!], \cdots, [\![s_{\ell-1}]\!])$ in the same bucket, perform the following check:
   (1) Let $[\![\mathbf{r+s}]\!]^A = [\![\mathbf{r}]\!]^A + [\![\mathbf{s}]\!]^A$.
   (2) Let $([\![c_0]\!], \cdots, [\![c_\ell]\!]) = \text{BitADDCarry}([\![r_0]\!], \cdots, [\![r_{\ell-1}]\!], [\![s_0]\!], \cdots, [\![s_{\ell-1}]\!])$ by using the remaining private quintuples to evaluate AND gates.
   (3) Call $\mathcal{F}_{\text{mixed.rB2A}}$ to convert $[\![c_\ell]\!] \mapsto [\![\psi(c_\ell)]\!]^A$.
   (4) Let $[\![\mathbf{c'}]\!]^A = [\![\mathbf{r+s}]\!]^A - 2^\ell \cdot [\![\psi(c_\ell)]\!]^A$. Open $\mathbf{c'}$ and $[\![c_0]\!], \cdots, [\![c_{\ell-1}]\!]$, and check that $\mathbf{c'}[h] = \sum_{i=0}^{\ell-1} \psi c_i[h] \cdot 2^i$.
8: If all the checks pass, output the first packed edaBit from each of the N buckets.

Figure 21: Protocol for RMFE-based private edaBit generation $\Pi_{\text{rEdabitPriv}}$.

Output: N packed daBit $\{([\![\mathbf{b_j}]\!]^A, [\![b_j]\!])\}_{j=1}^N$, where $\mathbf{b_j} \in \{0,1\}^k$ and $\mathbf{b_j}[i] = \psi(b_j)[i]$.
[Construct]

1: For each party P_i, it samples bits $b_{i,1}, \cdots, b_{i,(N+s)\cdot k} \in \{0,1\}$. P_i calls $\mathcal{F}_{\text{prep.rInput}}$ to obtain $[\![r_{i,j}]\!]$ where $r_{i,j} = \phi([b_{i,(j-1)k+1}, \cdots, b_{i,jk}])$, and calls $\mathcal{F}_{\text{prep.qInput}}$ to obtain $[\![\mathbf{r_{i,j}}]\!]^A$ where $\mathbf{r_{i,j}} = [b_{i,(j-1)k+1}, \cdots, b_{i,jk}]$, for $j = 1, \cdots, N+s$.
2: All parties compute $[\![\mathbf{r_j}]\!]^A = [\![\oplus_{i=1}^n \mathbf{r_{i,j}}]\!]^A$ and $[\![r_j]\!] = [\![\oplus_{i=1}^n r_{i,j}]\!]$. The $\oplus$ in the arithmetic world is computed as $a \oplus b = a+b-2ab$.

[Check]

3: Parties do the following s times:
   (1) Generate $(N+s) \cdot k$ fresh public random bits $a_1, \cdots, a_{(N+s)\cdot k} \in \{0,1\}$ and let $\mathbf{a_j} = [a_{(j-1)k+1}, \cdots, a_{jk}]$.
   (2) Compute $[\![\oplus_{j=1}^{N+s} \phi(\mathbf{a_j}) \cdot r_j]\!]$ and open it.
   (3) Compute $[\![\mathbf{r}]\!]^A = [\![\sum_{j=1}^{N+s} \mathbf{r_j} \odot \mathbf{a_j}]\!]^A$.
      - If $q = 2^\ell$, call $\mathbf{r'} = \text{open}([\![\mathbf{r} \cdot 2^{\ell-1}]\!]^A)$ and compute $\mathbf{r'}/2^{\ell-1} = (\mathbf{r} \cdot 2^{\ell-1} \mod 2^\ell)/2^{\ell-1} = \mathbf{r} \mod 2$.
      - If q is a prime, call $\mathbf{r'} = \text{open}([\![\mathbf{r}]\!]^A + 2 \cdot \sum_{i=0}^{s+1} [\![\mathbf{c_i}]\!]^A \cdot 2^i)$ with random bit vector $[\![\mathbf{c_i}]\!]^A$ and compute $\mathbf{r} \mod 2 = \mathbf{r'} \mod 2$.

   Abort if $\mathbf{r} \mod 2$ does not match the bits from the previous step.
4: Discard $([\![\mathbf{r_j}]\!]^A, [\![r_j]\!])$ for $j \in [N+1, N+s]$.
5: For $j \in [1, N]$, compute and open $[\![\mathbf{r_j} \odot (1 - \mathbf{r_j})]\!]^A$. Abort if any vector is not zero.

Figure 19: Protocol for RMFE-based daBit generation Π_{rDabit}.

D Mixed-circuit protocols

Packed daBit generation. To generate packed daBits, we vectorize the protocol from [19] for RMFE sharings in Figure 19. It securely implements the functionality $\mathcal{F}_{\text{mixed.rDabit}}$. The security proof is exactly the same as in [19] that works for any boolean domain. Nevertheless, the major cost in this protocol comes from the arithmetic multiplication, and RMFE's improvement to the boolean world is not obivious in the whole execution.

Boolean to arithmetic. To convert a RMFE sharing into a vector of arithmetic sharings, we use the protocol in Figure 20. It consumes a packed daBit. The flow is standard in the literature and similar protocols have been designed for traditional boolean sharings in previous works [18].

Private RMFE-based edaBit generation. Figure 21 gives the protocol that is adapted from [19] for RMFE sharings to implement the functionality $\mathcal{F}_{\text{mixed.rEdabitPriv}}$. It generates private packed edaBits whose underlying bits are known by one party. The protocol uses cut-and-choose and bucket sacrifice to ensure that P_i is producing valid packed edaBits. The security proof in [19] is general for any boolean domain and also applies to RMFE sharings. We avoid redundant description and refer the readers to [19] for details.

Table 9: Preprocessing material in the evaluation of SqueezeNet and ResNet (64 threads).

	SqueezeNet	ResNet	Commu. (KB / per op)		LAN Throughput (ops / per second)		WAN Throughput (ops / per second)	
			Tinier-LG	*Coral*-LG	Tinier-LG	*Coral*-LG	Tinier-LG	*Coral*-LG
dabit	5,349,231	10,613,224	7.58	7.77	95130	63976	3167	3081
strict edabit ($\ell = 9$)	1,000	0	56.4	40	17229	13309	439.6	616
strict edabit ($\ell = 10$)	0	2,048	58.6	40	16478	13300	416	605
strict edabit ($\ell = 12$)	2,649,992	20,121,577	63	41.2	15419	13089	386	588
strict edabit ($\ell = 31$)	5,349,231	10,613,224	106	46.4	10089	10489	229	522
strict edabit ($\ell = 41$)	5,349,231	10,613,224	128.6	49.6	8436	9644	188	490
strict edabit ($\ell = 71$)	2,650,992	20,123,625	196.4	58	5610	7363	133	433
bit triple	294,207,705	583,727,320	1.84	0.17	499390	681232	13167	136948

E More experiment details for large applications

The preprocessing material for non-linear computation in the large neural networks SqueezeNet and ResNet are listed in the 2nd and 3rd columns of Table 9. They are direct outputs from the compiler of MP-SPDZ. Based on the material requirement, we benchmark their communication and throughput under 64-thread environment, which we use to extrapolate the performance of non-linear computation for the two neural networks in Table 8.