FLUTE: Fast and Secure Lookup Table Evaluations (Full Version[⋆])

Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh and Hossein Yalame
Technical University of Darmstadt, Germany
Email: {`brueggemann,schneider,suresh,yalame`}`@encrypto.cs.tu-darmstadt.de`,
`hundt@stud.tu-darmstadt.de`

***Abstract*—The concept of using Lookup Tables (LUTs) instead of Boolean circuits is well-known and been widely applied in a variety of applications, including FPGAs, image processing, and database management systems. In cryptography, using such LUTs instead of conventional gates like AND and XOR results in more compact circuits and has been shown to substantially improve online performance when evaluated with secure multi-party computation. Several recent works on secure floating-point computations and privacy-preserving machine learning inference rely heavily on existing LUT techniques. However, they suffer from either large overhead in the setup phase or subpar online performance.**

We propose FLUTE, a novel protocol for secure LUT evaluation with good setup and online performance. In a two-party setting, we show that FLUTE matches or even outperforms the online performance of all prior approaches, while being competitive in terms of overall performance with the best prior LUT protocols. In addition, we provide an open-source implementation of FLUTE written in the Rust programming language, and implementations of the Boolean secure two-party computation protocols of ABY2.0 and silent OT. We find that FLUTE outperforms the state of the art by two orders of magnitude in the online phase while retaining similar overall communication.

1. Introduction

With data collection rising to unprecedented levels and consumers becoming more aware of and concerned about how their personal information is being used, effective privacy safeguards are more critical than ever. While using large amounts of user data creates new possibilities, such as in the health industry and for machine learning [64], doing so plainly compromises users' privacy to the point where it might even be against the law because of regulations like the General Data Protection Regulation (GDPR) or the California Consumer Privacy Act [42], [84]. Secure multi-party computation techniques (MPC) [39], [88] have demonstrated their ability to address this problem efficiently in a variety of real-world applications, including financial services [7], epidemiological modeling [41], privacy-preserving machine learning [13], [24], [43], [51], [55], and federated learning [12], [14], [61], [65]. MPC is a sophisticated cryptographic approach that allows a group of parties to compute a joint function on their private inputs while disclosing only the outputs and nothing else what cannot be derived from the outputs.

In the context of secure two-party computation (2PC) over boolean circuits with security against passive adversaries, which we explore in this paper, two of the most prominent protocols are Yao's garbled circuits (Yao) [89] and the Goldreich-Micali-Wigderson (GMW) protocol [39]. To maximize practical efficiency, these methods are typically divided into an input-independent setup phase and an input-dependent online phase [29], [86]. In detail, the approach enables the majority of the expensive cryptographic primitives, such as oblivious transfer (OT) [6], [18], [34], [74], to be completed in advance during the setup phase, leading to an extremely fast online phase once the inputs are available. Later research explored function-dependent preprocessing and demonstrated significant improvements by using knowledge of the underlying function to be evaluated [11], [68]. This is especially beneficial in practical scenarios such as machine-learning-as-a-service, where the same function is evaluated multiple times with different inputs.

GMW-based protocols require communication rounds that are linear in the multiplicative depth of the circuit yet are well-suited for high-throughput applications due to their minimal communication. In contrast, Yao-based protocols are preferred for low-latency solutions due to a constant number of rounds. The mutually orthogonal goals of these two approaches paved the way for Lookup Table-based (LUT) computation, which seeks to strike a balance between total communication and online round complexity [30], [33], [47]. For practical applications, including the most recent works on secure floating point evaluations [75] and privacy-preserving recurrent neural networks (RNN) inference [76], LUT-based techniques have been chosen over traditional GMW-style boolean circuit evaluations due to the improvements they bring. Concretely, SiRnn [76] uses LUTs to improve and accelerate the approximation of non-linear functions, such as sigmoid and tanh, that are essential in machine learning tasks.

Two LUT evaluation protocol variants, SP-LUT and OP-LUT, are proposed in [33], the state-of-the-art in the passive 2PC setting, to cater to different scenarios. While SP-LUT gets better total communication at the expense of higher on-

[⋆]*Please cite the conference version of this paper [20] that was published in IEEE Security & Privacy'23.*

line communication, OP-LUT has better online communication at the expense of very high total communication. However, the authors of [33] left open the problem of combining the best of both approaches, which is the starting point for our work. In particular, they pose the following question:

"Is it possible to combine the efficient setup phase of our SP-LUT approach with the efficient online phase of our OP-LUT approach and thereby obtain a protocol that achieves both, an efficient setup as well as an efficient online phase?"

We answer this question positively by providing a LUT evaluation protocol that is the first to offer the best of both worlds by being efficient during both the setup and online phases, leading to a significantly improved overall efficiency.

1.1. Our Contributions

In this work, we present FLUTE, an efficient protocol for evaluating lookup tables. While our technique is applicable in a variety of settings, we focus on the two-party setting (2PC) with semi-honest corruption. Our protocol employs function-dependent preprocessing to provide a highly efficient online phase in terms of both communication and rounds while maintaining overall communication at par with the state-of-the-art [33]. The following are our specific contributions:

1. ***A Novel Method for LUT Evaluation*** While all existing protocols for lookup table evaluation focus on perceiving a LUT as a table with δ inputs and σ outputs (cf. §3.1), we reimagine LUTs as a generalized notion of inner products over a boolean domain, a fresh perspective that results in significant performance improvements in our novel protocol FLUTE. In particular, the setup communication in our approach is *independent* of LUT outputs whereas the online communication is *independent* of LUT inputs. Moreover, evaluating an arbitrary δ-input LUT using FLUTE costs the same communication (both online and total) as a δ-input AND gate in the 2PC protocol ABY2.0 [68]. FLUTE uses the same sharing semantics as the underlying MPC protocol, allowing us to use our protocol in conjunction with the underlying secret sharing scheme at no additional cost, supporting mixed protocols for improved practical efficiency. Furthermore, our protocol's generic design allows for easy adaption to other settings, such as a larger number of parties and stronger security guarantees. This is in contrast to works such as [33], where changing the underlying 2PC semi-honest setting is not straightforward.

2. ***Extensive Analysis Incorporating State-of-the-Art Optimizations*** Prior works on LUT evaluation place a strong emphasis on either protocols with low online communication but high total communication (OTTT [47], improved upon by OP-LUT [33]), or protocols with low total communication but high online communication (SP-LUT [33]). FLUTE bridges this gap by offering a protocol that delivers the *best of both* worlds: Online communication similar to OP-LUT and total communication on par with SP-LUT.

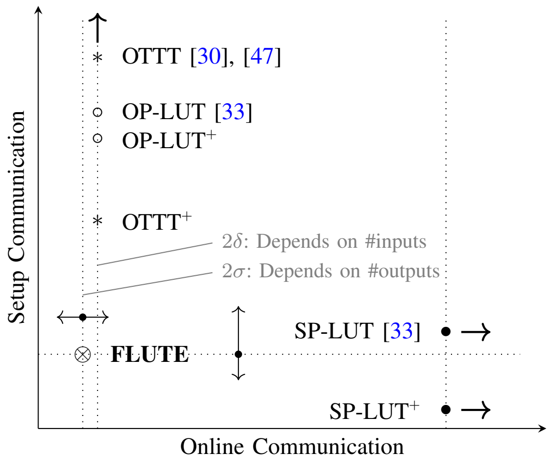


Figure 1: Comparison of setup and online communication between prior LUT protocols and FLUTE. $^+$ denotes the modification of prior LUTs that use silent OT [17]. The specific communication of FLUTE relative to other approaches depends on the LUT parameters: number of inputs δ and number of outputs σ. Exact costs depending on the given parameters are consolidated in Table 2.

Fig. 1 depicts an intuitive comparison of FLUTE with previous works in terms of setup and online communication costs. As indicated in the figure, FLUTE accomplishes online communication similar to OP-LUT and OTTT while preserving total communication comparable to SP-LUT, which is the state-of-the-art LUT protocol in terms of total communication. Fig. 1 also depicts the relative positions of prior approaches when their rather expensive oblivious transfer extension schemes in the style of IKNP [6], [46], [52] are replaced with the state-of-the-art optimizations using silent OT extension techniques [17], [18]. This comparison is backed by a comprehensive theoretical evaluation of all approaches in §4.1.

3. ***Open Source Implementation in Rust*** To support our theoretical findings, we provide an open source implementation of FLUTE written in the Rust programming language. Moreover, the implementation contains the Boolean secure two-party protocol in ABY2.0 [68] as well as the semi-honest silent OT extension approach in [17], both of which are implemented in Rust for the first time and are of independent interest. Our framework is open-sourced under the MIT License at https://encrypto.de/code/FLUTE. In contrast to various C/C++ MPC benchmarking frameworks in the literature, we opted for Rust as our implementation language because it provides great performance while maintaining memory safety and allowing for easy and safe parallelization, a.k.a. *fearless concurrency*. We emphasize that the high safety guarantees provided by Rust are especially important for creating secure protocols as the security also relies on a correctly working implementation.

4. ***Benchmarks Over a Wide Range of Circuits*** We illustrate the improved efficiency and practicality of our FLUTE protocol by evaluating a wide range of circuits including floating point operations. The appropriate LUT circuit representations are produced using a hardware synthesis

toolchain that combines Yosys [87], ABC [1], and Synopsis DC [2], [3] following the approach of [33]. §4 reports and evaluates our extensive evaluation results. When compared to the SP-LUT approach [33], FLUTE improves the online communication by more than $100\times$, while keeping the total communication overhead to less than 4% on average. Moreover, FLUTE could achieve a $3\times$ improvement in online communication when compared to the prior best LUT evaluation approaches with improved online phase, namely OTTT [30], [47] and OP-LUT [33].

2. Preliminaries

We consider two MPC servers, S_0 and S_1, that carry out the computation over a boolean ring, denoted by $\mathbb{Z}_2 = \{0,1\}$. As optional variant of our protocols, we also consider a commodity-based MPC setting in which an additional helper server $S_{\mathcal{H}}$ is used to improve protocol efficiency. The servers are connected by a bidirectional synchronous channel (e.g. instantiated via TLS over TCP/IP). We base our method on the two-party (2PC) protocol in ABY2.0 [68] and a short overview of the same is presented in §2.4. The servers, like in ABY2.0, perform a one-time key setup, denoted by the ideal functionality $\mathcal{F}_{\mathsf{key}}$, to enable non-interactive sampling of random values. The setup generates random keys for a pseudo-random function (PRF), which can be instantiated using, for example, AES in counter mode. For this, each server $S_i \in \{S_0, S_1\}$ selects a random PRF key share $\mathsf{K}^i \in \{0,1\}^\kappa$ and exchanges it, and the PRF key is defined as $\mathsf{K} = \mathsf{K}^0 \oplus \mathsf{K}^1$.

Function-dependent Preprocessing. Our protocol uses a *function-dependent* preprocessing [11], [22], [23], [68], in which the servers perform input-independent operations ahead of time in a *setup* phase and then use this data to support a fast *online* phase when the actual inputs are available. This approach is especially effective for improving the real-time performance of practical applications such as private machine learning inference and secure data analytics by providing higher throughput.

2.1. Threat Model

We assume that the two servers S_0 and S_1 are non-colluding and our protocol is secure against a single semi-honest (aka passive) corruption [25], [38]. Although it is a strong assumption to assume that corrupt parties cannot deviate from the protocol in a semi-honest security model, the development of protocols for this model is well justified for a number of reasons. For example, this can represent scenarios where you may trust your counterparty not to break the law in order to access your data, but you may not want or cannot give them the data outright [27]. This can also guard against system breaches caused by attackers attempting to inject undetectable malware into the system. These malwares will typically be passive and will attempt to steal as much data as possible from the machine (inluding incoming and outgoing communication). Moreover, this model acts as a stepping stone towards achieving stronger security guarantees. For instance, there exist several "compilers" that can increase the security of a semi-honest protocol with relatively little communication and computation overhead, e.g. [15], [19], [40], [57].

While we only discuss semi-honest corruption in this work, one key feature of our work, as will be made clear in section §3.2, is the design of the protocol, which allows it to be instantiated over various settings with different corruption thresholds and security assumptions.

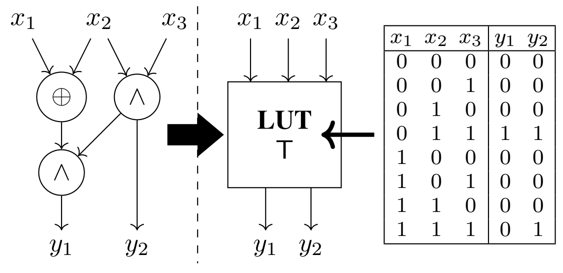
2.2. Oblivious Transfer

In our protocol, we use the oblivious transfer (OT) primitive [6], [34], [67], [74], which allows a sender to obliviously send a message of the receiver's choice to the receiver without the sender learning the choice and the receiver learning nothing about the set of other messages. Concretely, in a 1-out-of-N OT, the sender P_S inputs N messages $x_1, ..., x_N$, whereas the receiver P_R inputs an index $1 \leq c \leq N$, and P_R then only learns x_c while P_S learns nothing.

We use $\binom{N}{1}$-OT_ℓ^m to denote m instances of 1-out-of-N OT over ℓ bit inputs and $|\binom{N}{1}$-$\mathsf{OT}_\ell^m|$ denotes the required communication in bits. Moreover, $\binom{N}{1}$-rOT_ℓ^m denotes a variant named random OT where the messages $x_1, ..., x_N$ and the choice c are chosen at random by the underlying functionality. For $N = 2$, we use rOT_ℓ^m for brevity as this case occurs frequently in this work.

2.3. Lookup Tables

A lookup table (LUT) T is viewed as a multi-input multi-output boolean gate that maps $\delta \geq 2$ input bits to σ output bits according to an arbitrary boolean function $f : \{0,1\}^\delta \to \{0,1\}^\sigma$ [33], [47]. In FLUTE (cf. §3.2), we use the public encoding of LUT inputs and outputs, denoted as $\{\vec{\mathcal{E}}^u\}_{u\in[\delta]}$ and $\{\vec{\mathbf{y}}^w\}_{w\in[\sigma]}$, each having a bit size of 2^δ. For instance, in the example illustrated in Fig. 2, input $\vec{\mathcal{E}}^1$ corresponds to 00001111 and output $\vec{\mathbf{y}}^2$ corresponds to 00010001.



x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

Figure 2: Representation of a boolean circuit with $\delta = 3$ inputs and $\sigma = 2$ outputs as a δ-to-σ LUT T.

As discussed in [33], any boolean circuit, including complex functionalities like the AES S-Box, can be represented as a compact graph of interconnected LUTs and other linear gates using the above representation. Furthermore, the complexity of evaluating a LUT is determined solely by its size, i.e., the number of inputs and outputs, rather than by its internal logic.

2.4. Revisiting ABY2.0

We revisit the 2PC protocol in ABY2.0 [68] for secure evaluation of Boolean circuits. The protocol is run between two servers S_0 and S_1 and the operations are carried out over a boolean ring $\mathbb{Z}_2$. Given two boolean values x, y, $x \wedge y$ denotes the logical AND operation. $\overline{x}$ denotes the logical NOT operation.

2.4.1. Sharing Semantics. We utilize two sharing schemes over the boolean ring $\mathbb{Z}_2$.

$[\cdot]$-*sharing.* The $[\cdot]$-sharing of a boolean value $v \in \{0,1\}$ is an *XOR sharing* of v, i.e., S_0 holds $[v]_0$ and S_1 holds $[v]_1$ such that $[v]_0 \oplus [v]_1 = v$.

$\langle\cdot\rangle$-*sharing.* The $\langle\cdot\rangle$-sharing of $v \in \{0,1\}$ consists of two bits λ_v and $\mathsf{m}_v = v \oplus \lambda_v$, where λ_v is $[\cdot]$-shared among S_0, S_1 and m_v is known to both servers. We use λ_v^i to denote $[\lambda_v]_i$ for brevity.

Note that both secret sharing schemes are linear, i.e., given $\langle x\rangle, \langle y\rangle$ and constants a, b, c it holds that $\langle ax \oplus by \oplus c\rangle = a\langle x\rangle \oplus b\langle y\rangle \oplus c$. Servers S_0, S_1 compute this linear combination $\langle z\rangle = a\langle x\rangle \oplus b\langle y\rangle \oplus c$ without communication by setting $\mathsf{m}_z = a\mathsf{m}_x \oplus b\mathsf{m}_y \oplus c$ and $\lambda_z^i = a\lambda_x^i \oplus b\lambda_y^i$ for $i \in \{0,1\}$. This includes computing $\langle\cdot\rangle$-shares for the complement of bit x, which is denoted by $\overline{x}$ as $\langle\overline{x}\rangle = \langle 1 \oplus x\rangle$.

Given a set $\mathcal{Q}$, $\mathsf{m}_\mathcal{Q}$ denotes the AND of all the corresponding m bits and is given as $\mathsf{m}_\mathcal{Q} = \bigwedge_{q_i \in \mathcal{Q}} \mathsf{m}_{q_i}$. Similarly, $\lambda_\mathcal{Q} = \bigwedge_{q_i \in \mathcal{Q}} \lambda_{q_i}$ and the case when $\mathcal{Q} = \varnothing$ corresponds to $\mathsf{m}_\varnothing = \lambda_\varnothing = 1$.

2.4.2. AND operation. The idea behind the secure AND in ABY2.0 stems from the observation that for an AND gate with inputs $\langle x\rangle, \langle y\rangle$ and output $\langle z\rangle$ with $z = xy$ it holds that

$$\begin{aligned} z = xy &= (\mathsf{m}_x \oplus \lambda_x)(\mathsf{m}_y \oplus \lambda_y) \\ &= \mathsf{m}_x\mathsf{m}_y \oplus \lambda_x\mathsf{m}_y \oplus \lambda_y\mathsf{m}_x \oplus \lambda_x\lambda_y, \end{aligned} \tag{1}$$

where all terms except for $\lambda_x\lambda_y$ are linear combinations as $\mathsf{m}_x, \mathsf{m}_y$ are known to both servers. Now, a $[\cdot]$-sharing of the product $\lambda_{xy} := \lambda_x\lambda_y$ that is input-independent is computed in the setup phase using a black-box $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}(\lambda_x, \lambda_y)$ that can be instantiated using techniques like Oblivious Transfer (OT) [6], [18], [74] or Homomorphic Encryption (HE) [5], [66], [78]. In addition, servers non-interactively generate $[\cdot]$-shares of λ_z.

In the online phase, the servers locally compute a $[\cdot]$-sharing of $\mathsf{m}_z = z \oplus \lambda_z$ following equation (1) and exchange their shares to reconstruct m_z in a single round of interaction. Thus, the online communication is 2 bits while the setup communication is that of the $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}$ functionality (cf. §A).

2.4.3. Inner Product operation. Given two binary vectors of dimension d, $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, the inner product operation involves computing $z = \vec{\mathbf{x}} \odot \vec{\mathbf{y}} = \bigoplus_{i=1}^{\mathsf{d}} x_i y_i$. ABY2.0 extended the 2-input AND protocol to inner product computation with online communication independent of the vector dimension. At a high level, the idea is to execute the setup phase in accordance with the AND operation for each term of the form $x_i y_i$, but combine the communication for all the d terms in the online phase and communicate the combined value in a single shot. For more concrete details of the scheme, see [68].

2.4.4. Multi-Fan-In AND gates. In ABY2.0, the authors note that the protocol for a 2-input AND gate can be generalized to a k-input AND for arbitrary $k \geq 2$, while maintaining a fixed online communication. The crucial observation is that for $\langle\cdot\rangle$-sharing of inputs $\mathcal{I} = \{x_1, ..., x_k\}$ it holds that

$$z = \bigwedge_{i=1}^{k} x_i = \bigwedge_{i=1}^{k} (\mathsf{m}_{x_i} \oplus \lambda_{x_i}) = \bigoplus_{\mathcal{Q} \in 2^\mathcal{I}} (\mathsf{m}_\mathcal{Q} \wedge \lambda_{\mathcal{I}\setminus\mathcal{Q}}). \tag{2}$$

Unlike a 2-input AND gate, the servers need to compute $[\cdot]$-shares corresponding to $\lambda_\mathcal{Q}$ for all $\mathcal{Q} \in 2^\mathcal{I}$ and they achieve it using an instance of a $\mathcal{F}_{\mathsf{ANDM}}^{\mathsf{pre}}$ functionality. This makes the complexity of the setup phase grow exponential with the size of the input set $\mathcal{I}$. However, the online phase remains similar to that of a 2-input AND gate described in §2.4.2 and can be computed in a single round using 2 bits of communication.

2.5. Notations

Table 1 provides a concise description of the primary notations used in our paper.

TABLE 1: Notations used throughout this paper.

Notation	Description
MPC	Multi-party Computation; 2PC - two parties
S_0, S_1	2PC servers (non-colluding and semi-honest)
$S_\mathcal{H}$	Helper server (commodity-based MPC setting)
OT	Oblivious Transfer
$\binom{N}{1}$-OT_ℓ^m	m instances of 1-out-of-N OT over ℓ bits
$\binom{N}{1}$-rOT_ℓ^m	m instances of 1-out-of-N random OT over ℓ bits
$\lvert\cdot\rvert$	Cost function (communication in bits)
δ-to-σ T	Lookup Table (LUT) with δ inputs and σ outputs
$\overline{v}$	Complement of bit $v \in \{0,1\}$; $\overline{v} = 1 \oplus v$
$\vec{\mathbf{x}}$	Vector of dimension d; $\vec{\mathbf{x}} = (\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_\mathsf{d})$
$[v]$-sharing	XOR sharing; $S_i : [v]_i$ s.t. $[v]_0 \oplus [v]_1 = v$
$\langle v\rangle$-sharing	$S_i : (\mathsf{m}_v, \lambda_v^i)$ s.t. $v = \mathsf{m}_v \oplus \lambda_v^0 \oplus \lambda_v^1$
$2^\mathcal{I}$	Powerset of set $\mathcal{I}$; $x \in 2^\mathcal{I} \Leftrightarrow x \subseteq \mathcal{I}$

3. Evaluating LUTs

This section describes our protocol for evaluating LUTs with a highly efficient online phase. We start by taking a quick look at the existing methods for securely evaluating a δ-to-σ LUT.

3.1. Overview of Existing Approaches [33], [47]

In the context of 2PC, three prominent LUT evaluation procedures are available: i) One-Time Truth Tables (OTTT) [47], ii) Online-LUT (OP-LUT) [33], and iii) Setup-LUT (SP-LUT) [33].

3.1.1. One-Time Truth Tables. The idea behind the OTTT approach [47] is to generate an additive sharing of a LUT, rotated by a random additively-shared offset θ, among the two servers S_0 and S_1. Formally, given a LUT T, the setup phase comprises of generating two shares T^0 and T^1 such that for each entry i, it holds that $\mathsf{T}[i] = \mathsf{T}^0[i \oplus \theta] \oplus \mathsf{T}^1[i \oplus \theta]$. Moreover, S_0 holds (T^0, r) while S_1 holds (T^1, s) such that $\theta = r \oplus s$. Given a secret input $x = x_0 \oplus x_1$, in the online phase, the servers can generate an additive sharing of $\mathsf{T}[x]$ by reconstructing $x \oplus \theta$ and accessing that entry in their respective shares of T. Thus, the OTTT approach enables a fast online phase with 2δ bits of communication in a single round. The setup can be realized as in [30] by evaluating a Boolean circuit representing the table once for every possible input.

Lemma 1. (Communication) To evaluate a δ-to-σ LUT in the 2PC setting, the OTTT approach [30], [47] has communication of at most $(|\mathsf{MT}|+4)\cdot(\delta-1)\cdot 2^\delta\sigma$ bits[1] in the setup phase and 2δ bits in the online phase. Here, $|\mathsf{MT}|$ denotes the cost for generating a boolean multiplication triple.

3.1.2. Online-LUT. The OP-LUT approach [33] tries to reduce the cost of the OTTT setup phase by leveraging oblivious transfer (OT) instances while maintaining the same online phase as OTTT. In concrete terms, S_0 randomly samples its share of the LUT T^0 and then computes the other share T^1 for each potential offset value $s \in \{0,1\}^\delta$ that S_1 may choose. The servers then engage in a 1-out-of-2^δ OT with S_0 being the sender offering all possible shares T^1 as the inputs, and S_1 with its choice string s being the receiver. Each possible T^1 share is of size $2^\delta\sigma$ bits (2^δ rows, σ outputs each). The online phase of the protocol is the same as in the OTTT approach.

The OP-LUT approach, as mentioned in [33], can be considered as a natural generalization of the original GMW construction for evaluating 2-input AND gates using 1-out-of-4 OT [38, §7.3.3]. Also, [33] provided an optimization based on the observation that the receiver's choice string s is random which allows to have the OT protocol output it instead of taking it as an input. This allows to save a communication of δ bits in the OT protocol. We refer to [33] for more details.

Lemma 2. (Communication) To evaluate a δ-to-σ LUT in the 2PC setting, the OP-LUT approach [33] has communication of $|\binom{2^\delta}{1}\text{-}\mathsf{OT}^1_{2^\delta\sigma}| - \delta$ bits in the setup phase and 2δ bits in the online phase.

1. The number of AND gates in the Boolean circuit representation is bounded by $\delta - 1$.

3.1.3. Setup-LUT. This approach, SP-LUT [33], achieves better total communication than OTTT and OP-LUT at the expense of higher online communication. At a high level, the idea is to transfer all potential LUT outcomes in the online phase itself via a precomputed 1-out-of-N OT instance. In more detail, S_0 computes the LUT output for each possible set of S_1's input shares (a total of δ inputs resulting in 2^δ possible outputs) and prepares the share for S_1 for each of these possibilities. The servers then engage in a 1-out-of-2^δ OT instance, with S_1 as the receiver and, obtain its share for the LUT output.

The above approach allows the OTs to be precomputed [9], resulting in online communication of the 2^δ potential outputs for each of the σ LUT outputs. While this technique requires two online rounds (cf. §B), the amortized round complexity can be reduced to one for several use cases by swapping server roles in subsequent LUT evaluations [33].

Lemma 3. (Communication) To evaluate a δ-to-σ LUT in the 2PC setting, the SP-LUT approach [33] has communication of $|\binom{2^\delta}{1}\text{-}\mathsf{rOT}^1_\sigma|$ bits in the setup phase and $\delta + 2^\delta\sigma$ bits in the online phase.

Summary. [33] provides a comparison of the above three approaches with respect to a δ-to-σ LUT evaluation. They showed that OP-LUT improves OTTT setup communication for LUTs with small inputs ($\delta < 10$), whereas the SP-LUT approach outperforms the others in terms of overall communication. In terms of OTTT setup costs, [33] assumed a communication of 138 bits per AND gate due to its efficient multiplication triple generation (for more details, see [33, §III-E]). Recent advances in silent OT extension techniques [17], [18], [26], on the other hand, have decreased the cost of a boolean multiplication triple to less than 1 bit as described in §B. Thus, the OP-LUT approach outperforms the OTTT method only for $\delta < 4$ at this moment.

Looking ahead, we will compare with the best previous approach as a baseline, namely OTTT for the online cost, and SP-LUT for the total communication. Furthermore, we will use silent OT extension [17], [18] to improve communication during the setup phase. For a fair comparison, we instantiate the setup phases of the aforementioned approaches with silent OT extension as well. In particular, during the setup phase, we generate random 1-out-of-2 OTs and utilize them to construct boolean multiplication triples (MTs) as well as the 1-out-of-N OTs required by the preceding approaches. §B provides additional details.

3.2. The FLUTE Protocol

This section describes FLUTE, our approach for LUT evaluation with a fast online phase. We assume, as mentioned in §2.3, that the boolean circuit to be evaluated is represented as a compact graph of interconnected LUTs. The LUT protocol consists of creating $\langle\cdot\rangle$-shares for the input wires (cf. §2.4.1), evaluating the LUT gates using our FLUTE protocol along with local evaluation of linear gates such as XOR and NOT, and lastly retrieving the circuit output via output reconstruction. We concentrate on the LUT gate evaluation in this section since the other phases, as stated in §2.4, follow ABY2.0.

Sharing Semantics. The secret share for a δ-to-σ LUT T with inputs $(x_1, \dots, x_\delta)$ is defined as the set of $\langle\cdot\rangle$-shares corresponding to all its inputs, i.e., $\langle\mathsf{T}\rangle = \{\langle x_i\rangle\}_{i\in[\delta]}$.

FLUTE Overview. Without loss of generality, consider a δ-to-σ LUT T with an example illustration for $\delta = 3$ and $\sigma = 1$ given in Fig. 3. At a high level, the goal is to convert the LUT description into a boolean expression made up of basic logical gates (AND, XOR). The resulting expression can then be evaluated using the ABY2.0 protocol summarized in §2.4. In FLUTE, the conversion is carried out in three steps, and the details are given below.

x_1	x_2	x_3	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(a) LUT T with $\delta = 3$

LUT output z

=

$$(\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \vee(\overline{x_1} \wedge x_2 \wedge x_3) \vee(x_1 \wedge \overline{x_2} \wedge x_3)$$

(b) LUT Output Description

$$(\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \oplus(\overline{x_1} \wedge x_2 \wedge x_3) \oplus(x_1 \wedge \overline{x_2} \wedge x_3)$$

(c) OR to XOR Conversion

$$y = \vec{\mathcal{L}}^1 \odot \vec{\mathcal{L}}^2 \odot \vec{\mathcal{L}}^3 = \begin{pmatrix}\overline{x_1}\\ \overline{x_1}\\ x_1\end{pmatrix} \cdot \begin{pmatrix}\overline{x_2}\\ x_2\\ \overline{x_2}\end{pmatrix} \cdot \begin{pmatrix}\overline{x_3}\\ x_3\\ x_3\end{pmatrix}$$

(d) Output Computation

Figure 3: Example for transforming a $\delta = 3$-input single-output LUT to an instance of Multi-Fan-In Inner Product (cf. §3.2.1).

Step I - LUT Output Description (cf. Fig. 3b): Prior works, as mentioned in §3.1, securely compute LUTs by enumerating through all potential inputs (OTTT and OP-LUT) or all possible outputs (SP-LUT). After securely computing this information, the servers use their actual input shares to extract the right LUT output. We deviate from this approach and rather focus on deriving the LUT output from the inputs by first identifying only the relevant rows of the LUT that evaluate to 1 and then using a *full disjunctive normal form* representation to express the output as a function of the inputs.

Definition 1. (*DNF Representation* [31]) A boolean term $f(x_1, ..., x_\mu)$ is said to be in *full disjunctive normal form (DNF)*, if its a join of distinct terms of the form $x_1^{\epsilon_1} \wedge \dots \wedge x_\mu^{\epsilon_\mu}$. By definition, x^ϵ equals x if $\epsilon = 1$, and $\overline{x}$ if $\epsilon = 0$. In other words, $f(x_1, ..., x_\mu)$ can be expressed as $\bigvee_{j=1}^{k} \bigwedge_{i=1}^{\mu} \vec{\mathcal{L}}_j^i$, for some k, where for all i, j, $\vec{\mathcal{L}}_j^i \in \{x_i, \overline{x_i}\}$. Each $\vec{\mathcal{L}}^i$ can be seen of as a k-dimensional boolean vector.

To derive a DNF representation from the description of the LUT T, we consider only those rows of T whose output is 1. Let there be α such rows. Then, for each such row $j \in [\alpha]$, we build a term $\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i$ that only evaluates to 1 given the row's input assignment. In detail, we set $\vec{\mathcal{L}}_j^i = x_i$ if $x_i = 1$ in the assignment and set $\vec{\mathcal{L}}_j^i = \overline{x_i}$ if $x_i = 0$, for all $i \in [\delta]$. The output z is then represented as the OR of all such terms, i.e., $z = \bigvee_{j=1}^{\alpha} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i$.

In our example (Fig. 3a), the output is 1 only for the first, fourth and sixth rows of T. The inputs for the first row are $(x_1, x_2, x_3) = (0, 0, 0)$, hence the corresponding term is $\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}$. The remaining two terms are defined in the same way, and the output is expressed as the OR of these terms, as shown in Fig. 3b.

Given our goal of an efficient online phase, evaluating the DNF expression obtained above with a 2PC protocol such as ABY2.0 does not suffice. This is due to the DNF expression involving the evaluation of $\alpha\delta - 1$ non-linear operations (AND and OR combined) and a naive evaluation will require at least $\log(\alpha\delta)$ online rounds. This could be reduced to $\log_{\mathsf{n}}(\alpha\delta)$ using n-input AND gates, but at the expense of exponential communication in the setup phase.

Step II - OR to XOR Transformation (cf. Fig. 3c): In this step, we remove the OR operations required in the aforementioned DNF expression by making the following important observation: Given an assignment for the inputs $(x_1, \dots, x_\delta)$, two different terms of the form $\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i$ and $\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_{j'}^i$ can never both evaluate to 1 in the DNF expression obtained above. This is due to the fact that in a DNF (cf. Definition 1), all terms of the form $\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i$ are distinct, hence there will be at least one literal $\vec{\mathcal{L}}^i$ that differs for rows j and j'. Thus, in our case, it holds that

$$\bigvee_{j=1}^{\alpha} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i \equiv \bigoplus_{j=1}^{\alpha} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i. \quad (3)$$

Fig. 3c captures this transformation with respect to our example.

Step III - Output Computation (cf. Fig. 3d): This step efficiently computes the LUT output given in equation (3). For $\delta = 2$, the computation $\bigoplus_{j=1}^{\alpha} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i = \bigoplus_{j=1}^{\alpha} \vec{\mathcal{L}}_j^1 \wedge \vec{\mathcal{L}}_j^2$ is equivalent to an inner product computation over a boolean ring. On the other hand, when $\alpha = 1$, the same expression, $\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}^i$, simplifies to an instance of the multi-fan-in AND gate described in §2.4.4. Furthermore, utilizing the ABY2.0 protocol (cf. §2.4), evaluating an inner product as well as a multi-fan-in AND gate takes only a single round of interaction in the online phase and a communication of only 2 bits. So we combine these two primitives from ABY2.0 to get a protocol that evaluates expressions of the form $\bigoplus_j \bigwedge_i \vec{\mathbf{x}}_j^i$, which we call *Multi-Fan-In Inner Product* gates.

Let $\vec{\mathcal{L}}^i = (\vec{\mathcal{L}}_1^i, ..., \vec{\mathcal{L}}_\alpha^i)$ for $1 \leq i \leq \delta$. Then, using equation (3), the LUT output z can be written as

$$z = \bigoplus_{j=1}^{\alpha} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}_j^i = \bigodot_{i=1}^{\delta} \vec{\mathcal{L}}^i, \quad (4)$$

where $\bigodot$ denotes a multi-fan-in inner product operation and will be detailed in §3.2.1.

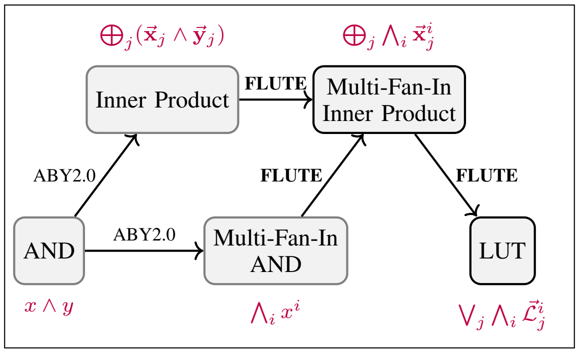

$\bigoplus_j(\vec{x}_j \wedge \vec{y}_j)$
$\bigoplus_j \bigwedge_i \vec{x}^i_j$
Inner Product
FLUTE
Multi-Fan-In Inner Product
ABY2.0
FLUTE
FLUTE
AND
ABY2.0
Multi-Fan-In AND
LUT
$x \wedge y$
$\bigwedge_i x^i$
$\bigvee_j \bigwedge_i \vec{\mathcal{L}}^i_j$

Figure 4: Roadmap of FLUTE protocol design starting with the known primitives in ABY2.0 [68]. Each node's functionality is provided alongside it.

Fig. 4 provides a roadmap of our protocol design. One advantage of our approach is that it is not limited to the 2PC setting discussed in this work, but may be applied to any MPC setting. However, as will be discussed later in §3.2.2, the specific costs incurred depend on the instantiations of these primitives in the respective setting. The details for realising the multi-fan-in inner product protocol using ABY2.0 (cf. §2.4) are provided next.

3.2.1. Multi-Fan-In Inner Product. Given a set of β input vectors $\mathcal{I} = \{\vec{x}^1, \ldots, \vec{x}^\beta\}$ of dimension d each, the goal is to compute $z = \vec{x}^1 \odot \cdots \odot \vec{x}^\beta = \bigoplus_{j=1}^{\mathsf{d}} (\bigwedge_{i=1}^{\beta} \vec{x}^i_j)$. Let $\mathcal{I}_j$ denote the set of values at the jth position in every vector $\vec{x}^i \in \mathcal{I}$, i.e., $\mathcal{I}_j = \{\vec{x}^1_j, \ldots, \vec{x}^\beta_j\}$. Then, following equation (2), we get

$$z = \bigoplus_{j=1}^{\mathsf{d}} \left(\bigwedge_{i=1}^{\beta} \vec{x}^i_j \right) = \bigoplus_{j=1}^{\mathsf{d}} \left(\bigwedge_{i=1}^{\beta} \left(\mathsf{m}_{\vec{x}^i_j} \oplus \lambda_{\vec{x}^i_j} \right) \right)$$
$$= \bigoplus_{j=1}^{\mathsf{d}} \left(\bigoplus_{\mathcal{Q}_j \in 2^{\mathcal{I}_j}} \left(\mathsf{m}_{\mathcal{Q}_j} \wedge \lambda_{\mathcal{I}_j \setminus \mathcal{Q}_j} \right) \right). \quad (5)$$

Similar to the case with a multi-input AND gate summarized in §2.4.4, the servers need to compute $[\cdot]$-shares corresponding to $\lambda_{\mathcal{Q}_j}$ for all $\mathcal{Q}_j \in 2^{\mathcal{I}_j}$ and $j \in [\mathsf{d}]$. Except for $\mathcal{Q}_j$ being a nullset or singleton, all cases involve communication across servers, which results in one instance of the $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ functionality being invoked for each $j \in [\mathsf{d}]$. However, the online communication can be made independent of both the number of inputs β and the vector dimension d, by combining the locally computed values in a manner similar to the inner product for fan-in 2 of ABY2.0 [68] as summarized in §2.4.3. This results in an online communication of just 2 bits and the formal protocol Π_{IP} is given Fig. 5.

The correctness of the Π_{IP} protocol is straightforward and follows from equation (5). In terms of setup phase communication, for a set $\mathcal{I}$ with β elements, the $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ functionality can be realised using $2^\beta - \beta - 1$ instances of $\mathcal{F}^{\mathsf{pre}}_{\mathsf{AND}}$, as detailed in §A. Furthermore, we use Beaver's multiplication [8] method to instantiate $\mathcal{F}^{\mathsf{pre}}_{\mathsf{AND}}$, which results in 4 bit communication among the servers given a boolean multiplication triple. §D provides a formal security proof for Π_{IP}.

Protocol $\Pi_{\mathsf{IP}}(\langle \vec{x}^1 \rangle, \ldots, \langle \vec{x}^\beta \rangle)$

Input: $\langle \cdot \rangle$-sharing of $\vec{x}^i$ for $i \in [\beta]$ and $|\vec{x}^i| = \mathsf{d}$.

Output: $\langle \cdot \rangle$-sharing of $z = \vec{x}^1 \odot \cdots \odot \vec{x}^\beta = \bigoplus_{j=1}^{\mathsf{d}} (\bigwedge_{i=1}^{\beta} \vec{x}^i_j)$.

Setup Phase:

1. Server S_i, for $i \in \{0,1\}$, samples random $\lambda^i_z \in \{0,1\}$.
2. Let $\mathcal{I}_j = \{\vec{x}^1_j, \ldots, \vec{x}^\beta_j\}$ for $j \in [\mathsf{d}]$. Then:
   Execute $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ to generate $[\lambda_{\mathcal{Q}_j}]$ for all $\mathcal{Q}_j \in 2^{\mathcal{I}_j}$.

Online Phase: Let $\mathcal{I}_j = \{\vec{x}^1_j, \ldots, \vec{x}^\beta_j\}$ for $j \in [\mathsf{d}]$.

1. Server S_i, for $i \in \{0,1\}$, locally computes
$$[v]_i = \bigoplus_{j=1}^{\mathsf{d}} \left(\bigoplus_{\substack{\mathcal{Q}_j \in 2^{\mathcal{I}_j}, \\ \mathcal{Q}_j \neq \mathcal{I}_j}} \left(\mathsf{m}_{\mathcal{Q}_j} \wedge \lambda^i_{\mathcal{I}_j \setminus \mathcal{Q}_j} \right) \right) \oplus \lambda^i_z.$$
2. Servers mutually exchange $[v]$ to obtain $v = [v]_0 \oplus [v]_1$.
3. Locally compute $\mathsf{m}_z = v \oplus \left(\bigoplus_{j=1}^{\mathsf{d}} \mathsf{m}_{\mathcal{I}_j} \right)$.

Figure 5: Multi-Fan-In Inner Product.

Lemma 4. (Communication) Protocol Π_{IP} (Fig. 5) incurs a communication of $(|\mathsf{MT}|+4) \cdot \mathsf{d} \cdot (2^\beta - \beta - 1)$ bits in the setup phase and 2 bits in the online phase to compute the inner product of a set of β d-dimensional vectors. Here, $|\mathsf{MT}|$ denotes the cost for generating a boolean multiplication triple.

3.2.2. LUT Evaluation Using FLUTE. Consider a δ-to-σ LUT T with inputs $(x_1, \ldots, x_\delta)$ and $\vec{y}^w \in \{0,1\}^{2^\delta}$ representing the encoding of the wth output, say $\vec{z}_w$, for $w \in [\sigma]$. This section shows how to securely evaluate the LUT using the multi-fan-in inner product protocol discussed above (cf. Π_{IP} in Fig. 5). Our method comprises of two steps, Input Preparation and Protocol Execution, as detailed next.

Step A - Input Preparation: This step prepares the input vectors for the inner product protocol using the LUT's inputs. Recall from the LUT output description discussed in §3.2 that the LUT output is solely dependent on rows where the corresponding bit in the output encoding is 1. In detail, for each LUT output $\vec{z}_w$, we must filter out the row k for which the respective encoding $\vec{y}^w_k = 1$. This can be easily achieved by incorporating the encoding $\vec{y}^w$ as another input to the inner product protocol Π_{IP} and considering all the rows of the LUT T. As a result, all the irrelevant rows of the LUT will be cancelled out as the corresponding entries in $\vec{y}^w$ are zeroes. Thus, following equation (4), the output $\vec{z}_w$ can be formally written as

$$\vec{\mathbf{z}}_w = \left(\bigoplus_{\substack{j \in 2^\delta, \\ \vec{\mathbf{y}}^w_j = 1}} \bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}^i_j \right) = \vec{\mathcal{L}}^1 \odot \ldots \odot \vec{\mathcal{L}}^\delta \odot \vec{\mathbf{y}}^w. \tag{6}$$

Since the vector $\vec{\mathbf{y}}^w$ is public, the aforementioned modification incurs no communication overhead over the Π_{IP} protocol with δ inputs, as detailed in the following step. Furthermore, since there is a specific pattern across every input column in a LUT, the modification simplifies the preparation of the input vectors $\vec{\mathcal{L}}^i$ for $i \in [\delta]$. The first column in T corresponding to the input x_1, for example, will have the first $2^{\delta-1}$ entries set to zero, followed by $2^{\delta-1}$ ones. As a result, the associated vector $\vec{\mathcal{L}}^1$ will be filled with $\overline{x_1}$ in the first half and x_1 in the rest. In general, the 2^δ entries in the ith vector $\vec{\mathcal{L}}^i$ will alternately be filled with blocks of $\overline{x_i}$ and x_i of size $2^{\delta-i}$, respectively. To formalize this, we use the LUT's input encoding information discussed in §2.3. The ith input vector $\vec{\mathcal{L}}^i$ is defined as

$$\vec{\mathcal{L}}^i_j = x_i \oplus (1 \oplus \vec{\mathcal{E}}^i_j), \quad \forall i \in [\delta], \forall j \in [2^\delta] \tag{7}$$

where $\vec{\mathcal{E}}^i$ denotes the input encoding vector for the ith input. Since the vector $\vec{\mathcal{E}}^i$ is public and our secret sharing scheme is linear, the computation of $\vec{\mathcal{L}}^i$ requires no communication among the servers.

Step B - Protocol Execution: Since each of the $\vec{\mathcal{L}}^i$ vectors has size 2^δ, evaluating Π_{IP} over these vectors naively will result in a factor of 2^δ in setup communication. We use the structure of the $\vec{\mathcal{L}}^i$ vectors prepared in the preceding step to get rid of this factor. In particular, each $\vec{\mathcal{L}}^i$ is made up of only the input x_i and its complement $\overline{x_i}$. Furthermore, as discussed in §2.4.1, $\langle \overline{x_i} \rangle$ is obtained by flipping only the masked value m_{x_i} and leaving the mask λ_{x_i} unchanged. In other words, for the case of LUT, we have $\lambda_{\vec{\mathcal{L}}^i_j} = \lambda_{x_i}$ for all $j \in [2^\delta]$ and $i \in [\delta]$. Thus, the preprocessing needs to be done only once for the set $\mathcal{I} = \{x_1, \ldots, x_\delta\}$, removing the factor 2^δ in setup communication discussed above. Using this observation, we incorporate the encoding vector $\vec{\mathbf{y}}^w$ to the computation with no additional communication:

$$\begin{aligned}
\vec{\mathbf{z}}_w &= \vec{\mathcal{L}}^1 \odot \ldots \odot \vec{\mathcal{L}}^\delta \odot \vec{\mathbf{y}}^w = \bigoplus_{j=1}^{2^\delta} \left(\left(\bigwedge_{i=1}^{\delta} \vec{\mathcal{L}}^i_j \right) \wedge \vec{\mathbf{y}}^w_j \right) \\
&= \bigoplus_{j=1}^{2^\delta} \left(\left(\bigwedge_{i=1}^{\delta} \left(\mathsf{m}_{\vec{\mathcal{L}}^i_j} \oplus \lambda_{\vec{\mathcal{L}}^i_j} \right) \right) \wedge \vec{\mathbf{y}}^w_j \right) \\
&= \bigoplus_{j=1}^{2^\delta} \left(\bigoplus_{\mathcal{Q}_j \in 2^{\mathcal{I}_j}} \left(\mathsf{m}_{\mathcal{Q}_j} \wedge \lambda_{\mathcal{I}_j \setminus \mathcal{Q}_j} \wedge \vec{\mathbf{y}}^w_j \right) \right) \\
&= \bigoplus_{\mathcal{Q} \in 2^{\mathcal{I}}} \left(\left(\bigoplus_{j=1}^{2^\delta} \left(\mathsf{m}_{\mathcal{Q}_j} \wedge \vec{\mathbf{y}}^w_j \right) \right) \wedge \lambda_{\mathcal{I} \setminus \mathcal{Q}} \right) \\
&= \bigoplus_{\mathcal{Q} \in 2^{\mathcal{I}}} \left((\vec{\mathsf{m}}_{\mathcal{Q}} \odot \vec{\mathbf{y}}^w) \wedge \lambda_{\mathcal{I} \setminus \mathcal{Q}} \right),
\end{aligned} \tag{8}$$

where $\mathcal{Q}_j$ for some $\mathcal{Q} \in \mathcal{I}$ denotes replacing each x_i in $\mathcal{Q}$ by $\vec{\mathcal{L}}^i_j$, and $\vec{\mathsf{m}}_{\mathcal{Q}}$ denotes a vector of dimension 2^δ with elements $\mathsf{m}_{\mathcal{Q}_j}$ for every $\mathcal{Q}_j \in 2^{\mathcal{I}_j}$.

The computation of $\vec{\mathbf{z}}_w$ in equation (8) resembles the Π_{IP} protocol discussed in §3.2.1 and the formal protocol Π_{LUT} is given in Fig. 6.

Protocol $\Pi_{\mathsf{LUT}}((\langle x_1 \rangle, ..., \langle x_\delta \rangle), \mathsf{T})$

Input: LUT T for a function $f : \{0,1\}^\delta \to \{0,1\}^\sigma$, with inputs $\langle x_k \rangle$ and input encoding $\vec{\mathcal{E}}^k \in \{0,1\}^{2^\delta}$, for $k \in [\delta]$, and output encoding $\vec{\mathbf{y}}^w \in \{0,1\}^{2^\delta}$, for $w \in [\sigma]$.

Output: $\langle \vec{\mathbf{z}} \rangle$, where $\vec{\mathbf{z}} = (\vec{\mathbf{z}}_1, ..., \vec{\mathbf{z}}_\sigma) = \mathsf{T}[x_1, ..., x_\delta]$.

Setup Phase:

1. Server S_i, for $i \in \{0,1\}$ samples random $\lambda^i_{\vec{\mathbf{z}}} \in \{0,1\}^\sigma$.
2. Let $\mathcal{I} = \{x_1, \ldots, x_\delta\}$. Then:
   – Execute $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ to generate $[\lambda_{\mathcal{Q}}]$ for all $\mathcal{Q} \in 2^{\mathcal{I}}$.

Online Phase:

1. *Input Preparation:*
   – Locally set $\vec{\mathcal{L}}^i_j = x_i \oplus (1 \oplus \vec{\mathcal{E}}^i_j), \quad \forall i \in [\delta], \forall j \in [2^\delta]$.
   – Locally set $\mathcal{I}_j = \{\vec{\mathcal{L}}^1_j, \ldots, \vec{\mathcal{L}}^\delta_j\}, \quad \forall j \in [2^\delta]$.
2. Server S_i, for $i \in \{0,1\}$ and $w \in [\sigma]$, locally computes
$$[\vec{\mathbf{v}}_w]_i = \bigoplus_{\mathcal{Q} \in 2^{\mathcal{I}}, \mathcal{Q} \neq \mathcal{I}} \left((\vec{\mathsf{m}}_{\mathcal{Q}} \odot \vec{\mathbf{y}}^w) \wedge \lambda_{\mathcal{I} \setminus \mathcal{Q}} \right) \oplus \lambda^i_{\vec{\mathbf{z}}_w}.$$
3. Servers mutually exchange $[\vec{\mathbf{v}}]$ to obtain $\vec{\mathbf{v}} = [\vec{\mathbf{v}}]_0 \oplus [\vec{\mathbf{v}}]_1$.
4. Locally compute $\mathsf{m}_{\vec{\mathbf{z}}_w} = \vec{\mathbf{v}}_w \oplus (\vec{\mathsf{m}}_{\mathcal{I}} \odot \vec{\mathbf{y}}^w)$.

Figure 6: FLUTE Lookup Table Evaluation

The correctness of the Π_{LUT} protocol follows from equation (8) and the security is similar to the Π_{IP} protocol in Fig. 5. One key difference between Π_{LUT} and Π_{IP} is that in Π_{LUT}, multiple output wires use the same set of preprocessing materials generated using $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$. Despite this, privacy is still maintained as each message $[\vec{\mathbf{v}}_w]_i$ exchanged in the online phase contains a randomly sampled mask $\lambda^i_{\vec{\mathbf{z}}_w}$, not known to either of the servers. The formal details of instantiating $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ using Beaver's multiplication method [8] are given in §A and the security details are elaborated in §D.

Concerning communication, as previously mentioned in the protocol execution step, preprocessing is only required once for a set of δ elements, necessitating a single call to $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$, which can be implemented using $2^\delta - \delta - 1$ calls to $\mathcal{F}^{\mathsf{pre}}_{\mathsf{AND}}$ (cf. §A). The online phase requires a communication of 2 bits per output wire similar to ABY2.0 [68].

Lemma 5. (Communication) To evaluate a δ-to-σ LUT in the 2PC setting, FLUTE (Π_{LUT} in Fig. 6) has communication of $(|\mathsf{MT}| + 4) \cdot (2^\delta - \delta - 1)$ bits in the setup phase and 2σ bits in the online phase. Here, $|\mathsf{MT}|$ denotes the cost for generating a boolean multiplication triple.

Besides the 2PC setting, we also explore the potential benefits that an untrusted helper server $S_{\mathcal{H}}$ can provide to the setup phase. At a high level, $S_{\mathcal{H}}$ will be given the λ_{x_i}-shares that correspond to all the inputs $\langle x_i \rangle$ for $i \in [\delta]$.

This allows $S_{\mathcal{H}}$ to perform the computation associated with $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ locally. $S_{\mathcal{H}}$ will then generate a $[\cdot]$-share of the values it computed, resulting in communication of $(2^\delta - \delta - 1)$ bits in the setup phase. Details are deferred to §C due to space constraints.

4. Evaluation

In this section, we compare and evaluate our FLUTE protocol against the existing LUT evaluation approaches OTTT [30], [47], OP-LUT [33], and SP-LUT [33], which were summarized in §3.1. While §4.1 deals with an analytical assessment of a single LUT instance's communication, §4.2 includes both theoretical and practical evaluation results for a range of LUT circuits. Additional benchmark results are given in §E. While each functionality might theoretically be represented by a single large LUT, as discussed in [33], this would significantly increase communication. As a result, in this work, we consider δ-to-σ LUTs with up to $\delta = 8$ inputs and $\sigma = 8$ outputs in accordance with [33]. Furthermore, for the evaluation, we assume that the communication for one instance of a random 1-out-of-2 OT, rOT^1_ℓ, is ≈ 0.118 bits, based on our implementation of silent OT extension in [17] for a batch size of 10^7 with 128-bit security.

4.1. Analytical Communication Costs

This section focuses on the setup and online communication of a δ-to-σ LUT for varying choices of δ and σ. For a fair comparison, communication for OT instances is provided utilizing Beaver's OT precomputation technique [9], which generates a random OT first, followed by a standard conversion to obtain OT over the actual inputs (cf. §B.2). Table 2 provides a summary of both the setup and online communication of the different approaches. The factor $(|\mathsf{MT}| + 4)$ in the setup communication of both OTTT and FLUTE stems from the OT-based instantiation of their setup phase, as discussed in §A. However, their setup phase might be realized using alternative techniques such as homomorphic encryption [68], providing greater flexibility than the OP-LUT and SP-LUT approaches, which are designed to work using only OT instances. In contrast to OTTT and OP-LUT, the online communication of FLUTE depends solely on the number of LUT outputs σ rather than the inputs δ. As will be seen in §4.2 for the case of real circuits, often $\sigma \leq \delta$, which leads to improved online communication. Fig. 14 in §E depicts the protocols' online communication for different LUT sizes.

Since [33] evaluated OTTT, OP-LUT, and SP-LUT approaches using their optimized 1-out-of-N OT extension protocol, the impact of the recent silent OT optimizations [17] on these approaches is unclear. For example, when using the optimized protocols from [33], one instance of 1-out-of-2 OT has a communication of 138 bits, whereas silent OT reduces it to only 4.2 bits. As a result, we analyze these protocols by substituting silent OT for OT instances in [33], and the results are shown in Table 2 with a superscript "+". Note that the online phase of these approaches remains unchanged. While [33] showed that OP-LUT outperforms OTTT in terms of setup communication for $\delta \leq 8$, this argument does not hold true when comparing the silent OT versions. Concretely, for $\delta \geq 4$, OTTT$^+$ outperforms OP-LUT$^+$ with respect to setup communication, while its cost is at most 6% worse for smaller choices of δ. We infer that efficient OT protocols will render OP-LUT obsolete.

TABLE 2: Setup and online communication of existing LUT evaluation approaches [30], [33], [47], and FLUTE for LUTs with δ inputs and σ outputs. We use silent OT cost as $|\mathsf{rOT}^1_\ell| \approx 0.118$ bits [17].

Protocol	Setup	Online
Generic cost (2PC)		
OTTT [30], [47]	$\leq (\lvert\mathsf{MT}\rvert + 4)(\delta - 1)2^\delta\sigma$	2δ
OP-LUT [33]	$\lvert\binom{2^\delta}{1}\text{-}\mathsf{rOT}^1_{2^\delta\sigma}\rvert + 2^{2\delta}\sigma$	2δ
SP-LUT [33]	$\lvert\binom{2^\delta}{1}\text{-}\mathsf{rOT}^1_{\sigma}\rvert$	$2^\delta\sigma + \delta$
FLUTE (this work)	$(\lvert\mathsf{MT}\rvert + 4)(2^\delta - \delta - 1)$	2σ
Concrete cost in bits as in [33] (2PC)		
OTTT [30], [47]	$\leq 138(\delta - 1)2^\delta\sigma$	2δ
OP-LUT [33]	$\geq 190 + 2^{2\delta}\sigma$	2δ
SP-LUT [33]	≥ 190	$2^\delta\sigma + \delta$
Concrete cost in bits using silent OT [17] (2PC)		
OTTT$^+$	$\leq 4.236(\delta - 1)2^\delta\sigma$	2δ
OP-LUT$^+$	$0.118\delta + 2^{2\delta}\sigma$	2δ
SP-LUT$^+$	0.118δ	$2^\delta\sigma + \delta$
FLUTE (this work)	$4.236(2^\delta - \delta - 1)$	2σ
Concrete cost in bits using a helper server $S_{\mathcal{H}}$		
OTTT$^{\mathcal{H}}$	$2^\delta\sigma$	2δ
FLUTE$^{\mathcal{H}}$ (this work)	$2^\delta - \delta - 1$	2σ

In terms of total communication, SP-LUT$^+$ is a close competitor to FLUTE. Since a precise comparison of the total communication of these two approaches is difficult due to the reliance on the choice of δ and σ, we plot the total communication versus δ for various σ in Fig. 7. We infer that for larger values of σ, FLUTE will start to outperform SP-LUT$^+$ in terms of total communication. This is because the expensive component of FLUTE's total communication only depends on δ, whereas SP-LUT$^+$ depends on both δ and σ.

Table 3 summarizes the improvement of FLUTE over SP-LUT$^+$ with respect to total communication. We observe that for $\sigma \geq 5$, FLUTE outperforms SP-LUT$^+$, while for smaller values of σ, it does so only for low values of δ. A similar comparison of improvements in the online communication is provided in Table 7 in §E.

In terms of total communication, we conclude that FLUTE not only outperforms previous LUT protocols with improved online phase, but also outperforms SP-LUT$^+$, the state-of-the-art in total communication, for different LUT sizes.

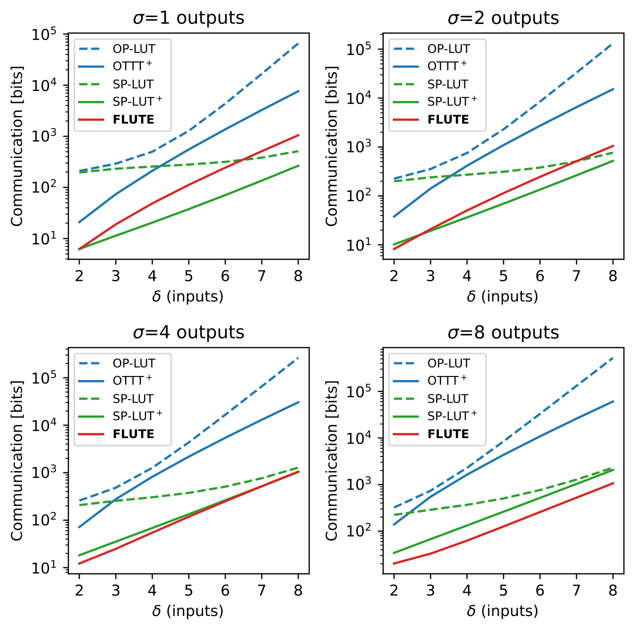


Figure 7: Total communication for different LUT sizes with $2 \le \delta \le 8$ inputs and $\sigma \in \{1, 2, 4, 8\}$ outputs.

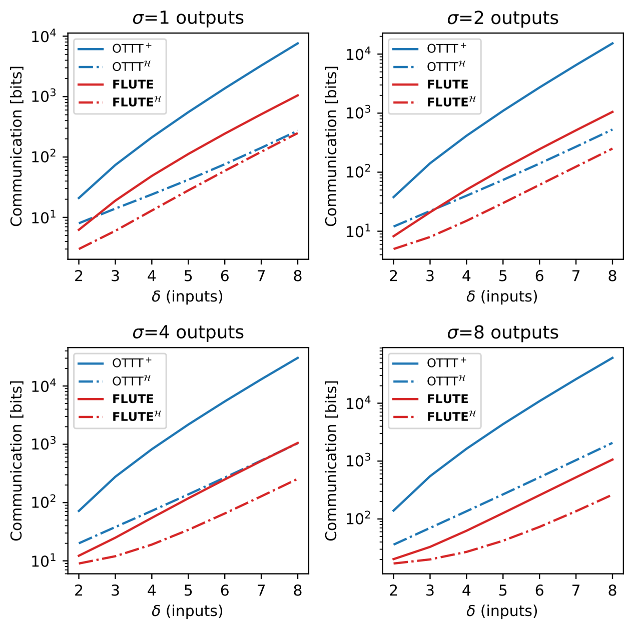


Figure 8: Total communication for different LUT sizes with $2 \le \delta \le 8$ inputs and $\sigma \in \{1, 2, 4, 8\}$ outputs using a helper server $S_{\mathcal{H}}$ (cf. §C).

TABLE 3: Improvement factor of total communication of FLUTE over SP-LUT$^+$ when evaluating one δ-input σ-output LUT. Highlighted cells correspond to the configurations where FLUTE outperforms SP-LUT$^+$.

δ \ σ	1	2	3	4	5	6	7	8
2	1.00	1.24	1.39	1.49	1.56	1.62	1.66	1.69
3	0.60	0.92	1.19	1.42	1.61	1.77	1.92	2.04
4	0.42	0.72	1.00	1.25	1.49	1.71	1.92	2.12
5	0.34	0.61	0.87	1.13	1.38	1.62	1.85	2.07
6	0.29	0.55	0.80	1.05	1.30	1.54	1.78	2.01
7	0.27	0.51	0.76	1.01	1.25	1.49	1.73	1.97
8	0.25	0.50	0.74	0.98	1.22	1.46	1.70	1.94

Optimizations using Helper Server. We close our analytical evaluation by reviewing the improvement that can be achieved by using an untrusted helper in the setup phase as described in §C. This setting, for the case of OTTT and FLUTE, is shown in Table 2 with a superscript "$\mathcal{H}$".

Fig. 8 provides a comparison of OTTT$^+$, OTTT$^{\mathcal{H}}$, FLUTE and FLUTE$^{\mathcal{H}}$ in terms of total communication. We observe that a helper server yields significant improvements in communication for both OTTT and FLUTE. Moreover, FLUTE$^{\mathcal{H}}$ has better total communication than OTTT$^{\mathcal{H}}$, even by a large margin except for high δ and low σ.

4.2. Experimental Evaluation for Real Circuits

In this section, we evaluate the performance of FLUTE on LUT representations for basic and complex operations. In particular, we use the following circuits: addition (ripple-carry Add-RC [48] and Ladner-Fischer Add-LF [56]), multiplication (ripple-carry Mul-RC [48] and Ladner-Fischer Mul-LF [82]), tree-based greater-than GT-Tree [37], the AES S-Box [16], and floating point operations [32]. The circuits are generated using a hardware synthesis toolchain similar to the one used in [33]. We use Yosys [87] as an open-source framework for front-end processing of our Verilog HDL to map them into a network of low-level logic operations in an intermediate format. Then, the ABC tool [1] is used to structure this network into a Directed Acyclic Graph (DAG) and maps it into LUTs in a depth-optimized fashion. To generate LUTs of more complex functionalities, such as trigonometric and floating point functions, we use the hardware Intellectual Property (IP) libraries in the Synopsys Design Compiler (DC) [2], [3].

Table 4 shows the distribution of LUT sizes with respect to the number of inputs δ for the circuits that we consider in this work. Following [33], we restrict the LUT sizes up to 8 ($1 \le \delta, \sigma \le 8$) in the hardware synthesis tool. Moreover, we apply the post-processing optimization from [33] that combines smaller LUTs that are compatible with each other into larger LUTs. By doing so, we are able to reduce the costs that would otherwise be incurred when recomputing the same set of values. Notably, the Add-RC circuit comprises 17 LUTs, among which 13 possess no less than 7 inputs. In contrast, the FP-DIV circuit involves 964 LUTs, with 831 containing at least 7 inputs. The AES S-box, in comparison, employs a 8-to-8 LUT as its primary design element.

Fig. 9 depicts an empirical evaluation of FLUTE as well as the OTTT and SP-LUT approaches with silent OT optimization. The online and total communication of these approaches are computed by parsing the LUT architecture from the LUT description files. Furthermore, we include the total communication for the constant round solution of Yao's garbled circuits protocol (Yao) [89], which incorporates the

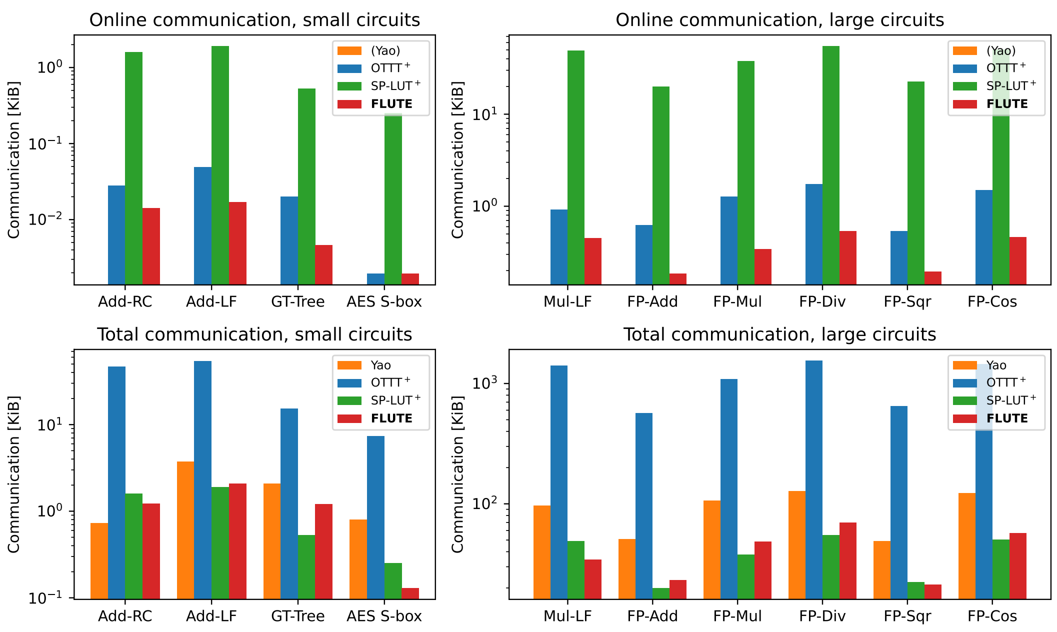


Figure 9: Online and total communication using Yao's garbled circuits [89] using three-halves garbling [80] (Yao), OTTT [30], [47] with silent OT [17] (OTTT$^+$), SP-LUT [33] with silent OT [17] (SP-LUT$^+$) and FLUTE. We omit Yao in the online communication for a fair comparison to secret-shared protocols, where the inputs are assumed to be secret-shared.

TABLE 4: Distribution of LUT sizes for various circuits in FLUTE. #LUTδ denotes the number of δ-to-σ LUTs with $1 \leq \sigma \leq 8$.

Circuit	#LUT2	#LUT3	#LUT4	#LUT5	#LUT6	#LUT7	#LUT8
Basic Functionalities							
Add-RC	–	4	–	–	–	1	12
Sub-RC	–	4	–	1	–	–	11
Mul-RC	–	–	2	4	211	53	436
Add-LF	–	–	–	2	1	7	17
Sub-LF	–	–	–	2	1	7	17
Mul-LF	–	–	2	15	51	111	324
GT-Tree	–	1	–	–	–	1	9
AES S-box	–	–	–	–	–	–	1
Floating-Point Functionalities							
FP-Add	–	–	5	10	16	97	213
FP-Sub	1	2	5	8	19	88	205
FP-Mul	1	–	4	11	31	148	485
FP-Div	–	6	14	26	86	267	564
FP-Sqr	1	–	–	5	18	53	210
FP-Sqrt	2	5	7	13	58	131	285
FP-Sin	–	2	8	17	66	195	551
FP-Cos	2	1	6	23	83	196	515

recent three-halves garbling optimization of [80]. Regarding the online communication, we find that FLUTE not only outperforms SP-LUT$^+$ by a factor of over $99\times$ in all cases, but also offers significant advantages over OTTT$^+$, which it outperforms by a factor of $1.88\times$ up to $4.32\times$. The AES S-box is the only exception, as it is implemented by a single LUT with 8 inputs and outputs, producing the same online communication as OTTT$^+$. Recall that the online communication of OTTT$^+$ is twice the number of inputs for each LUT evaluation, while the communication of FLUTE is twice the number of outputs. As a result, our findings indicate the number of LUT outputs σ is typically less than the number of inputs δ. In terms of total communication, FLUTE outperforms OTTT$^+$ in all cases and by a factor with a geometric mean[2] of $27.61\times$ across all circuits. In some cases, compared to SP-LUT$^+$, FLUTE improves between $1.43\times - 1.94\times$ in total communication, however in others, SP-LUT$^+$ has up to $2.33\times$ better communication, with the geometric mean being 4% less communication in favor of SP-LUT$^+$. In comparison, the constant round Yao only provides lower total communication in some of the smaller circuits while being outperformed by FLUTE by a factor ranging from $1.71\times - 6.14\times$ in the remainder.

TABLE 5: Online communication rounds of FLUTE, OTTT, SP-LUT and ABY2.0 for floating point operations.

Circuit	FLUTE/OTTT	SP-LUT	ABY2.0
FP-Add	18	19	59
FP-Mul	16	17	47
FP-Div	78	79	296
FP-Sqr	11	12	41
FP-Cos	25	26	98

Recall that, in addition to reducing the online communication, the goal of using LUTs is to minimize the number of online communication rounds. Table 5 provides an overview of the online rounds for FLUTE and prior LUT approaches along with a standard boolean circuit evaluation[3] using the ABY2.0 protocol [68]. We only consider the larger floating-point circuits, because smaller circuits resulted in mostly

2. Using the arithmetic mean would be ineligible for relative performance [35].

3. We used publicly available .aby and .bristol circuits that only had 2-input AND gates and no multi-input AND gates.

trivial round complexities. We find that the round complexity is reduced by a factor of up to $3.92\times$, especially for large circuits. We do not compare to Yao since we do not consider the input sharing phase, and Yao does not require any further online interactions.

Rust Implementation. We proceed with benchmarks of our FLUTE implementation developed in the Rust [4] language to analyze how the actual performance of FLUTE compares to the theoretical baseline. Rust combines high performance similar to C++ with memory safety and thread safety, all aspects that are of special interest for cryptographic implementations that should not only be fast, but also minimize the risk of code vulnerabilities. Along the way, we implemented the 2PC semi-honest boolean protocols in ABY2.0 [68], as well as silent OT [17], both of which are their first implementations in Rust that are of independent interest. Our implementation of silent OT [17] is based on the C++-based code available in the libOTe [71] library. We used a batch size of 10^7, compression factor of 2 and the protocols are implemented with 128-bit security.

Benchmarking Environment. We run the benchmarks on a server equipped with a 16-core Intel Core i7-4790 CPU at 3.6 GHz and 32 GB of RAM operated at 2400 MHz. Realistic network behaviour is simulated using the tools `tc` (traffic control) and `NetEm`. The benchmarks are run in a LAN setting with 10 GBit/s bandwidth and 1 ms round-trip time (RTT) and a WAN setting at 100 MBit/s bandwidth and 100 ms RTT. For our benchmarks, we use the Rust stable toolchain v1.65.0.

TABLE 6: Theoretical and experimental evaluation of FLUTE in terms of total communication in KiB (first online, then total) over batch sizes $\{1, 1000\}$.

Circuit	Theoretical		Batch 1		Batch 1000	
Add-RC	0.014	1.22	0.318	1.80	0.015	1.22
Add-LF	0.017	2.09	0.119	2.45	0.017	2.09
GT-Tree	0.005	1.22	0.107	1.59	0.005	1.22
S-box	0.002	0.13	0.035	0.42	0.002	0.13
Mul-LF	0.452	34.29	0.723	34.86	0.453	34.29
FP-Add	0.186	23.28	0.798	24.19	0.187	23.28
FP-Mul	0.340	48.74	0.936	49.64	0.340	48.74
FP-Div	0.534	69.51	3.214	72.49	0.537	69.52
FP-Sqr	0.194	21.39	0.591	22.08	0.194	21.39
FP-Cos	0.462	57.15	1.303	58.30	0.463	57.15

Table 6 provides a comparison of the actual communication obtained from our implementation and the theoretical baseline for various circuits for batch sizes $\{1,1000\}$. Without batching, our approach incurs some overhead over the theoretical baseline, particularly during the online phase. This is mostly due to the relatively low number of payload bits sent and received, which increases the impact of serialization and paddings. Increasing batch sizes dramatically reduces such overheads to the point that the measured values almost identically match the theoretical baseline for batch size 1000.

In Fig. 10, we compare the run time of FLUTE with the 2PC baseline in ABY2.0 [68] with silent OT, denoted by ABY2.0$^+$, over a LAN and WAN setting. Table 9 in §E provides the concrete results.

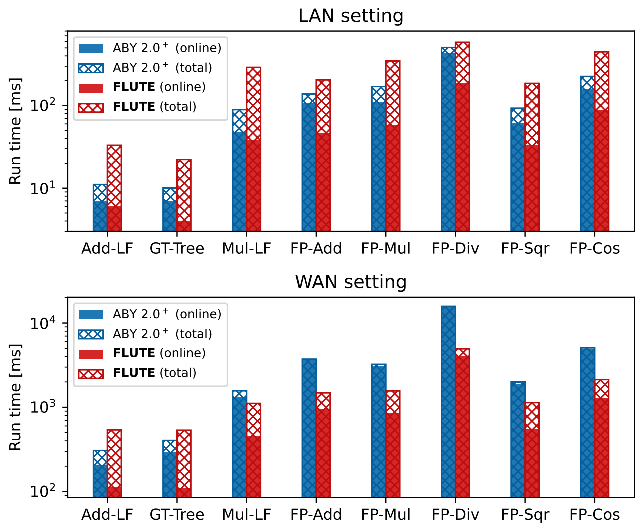


Figure 10: Run time of our implementations of FLUTE and ABY2.0 [68] with silent OT [17] (ABY2.0$^+$) for various circuits and batch size 1.

In the LAN setting, when compared to ABY2.0$^+$, FLUTE improves the online run time by a factor ranging from $1.17\times$ for small circuits and $1.27\times$ for large circuits up to $2.33\times$. The total run time is between 14% and 66% less for ABY2.0$^+$ owing to the expensive computation in FLUTE's setup and the time for computation being the dominant factor over communication in a LAN. However, for large circuits, the gap narrows when the online phase begins to dominate the setup phase. Over a WAN, the improvement in online run time of FLUTE over ABY2.0$^+$ rises to a factor ranging between $1.82\times$ and $3.82\times$. In addition, the increased RTT takes a toll on ABY2.0$^+$, so that even the total run time of FLUTE outperforms ABY2.0$^+$ in all large circuits by a factor ranging from $1.41\times$ up to $3.19\times$. This demonstrates that FLUTE can not only be used to improve the online time of the state-of-the-art 2PC protocols, but can also greatly outperform them in both online and total run time depending on the network setting.

5. Related Work

This section provides a concise summary of the important related work, with a focus on the 2PC semi-honest setting considered in this work.

Two-Party Computation (2PC). The arguably oldest 2PC protocol is Yao's garbled circuits protocol (Yao) [89] which has been subject to ongoing optimizations, the most recent one given by [80]. Its key aspect is that it essentially splits the evaluation of a binary circuit into *garbling* where one party, the garbler sends a so called garbled circuit to the other party, the evaluator. Further communication is only required to let the evaluator obtain encrypted versions of both parties' inputs. Another commonly used technique is secret sharing (SS) where each input is shared between both parties that then can locally evaluate linear gates while non-linear gates require further interaction. One well-known example is the GMW protocol [39].

Preprocessing Model. Yao naturally splits the computation into an input-independent setup phase and an input-dependent online phase. For SS-based MPC, a similar segmentation that was initiated by [8], [50], [82] lets the parties generate correlated randomness in the setup phase which later speeds up the online phase regarding communication, interactive rounds and hence also run time. These works started a rich line of work on increasingly efficient MPC protocols in the preprocessing model [23], [33], [53], [54], [62], [68], [70], [75], [76]. Some works, most notably LUT-based protocols [33], even have a function-dependent setup phase, i.e., they assume the parties already know the function during setup [11], [58], [68], [86]. ABY2.0 [68] is one such work that for the binary domain also implements multi-input AND gates at the same online cost as standard two-input ANDs.

Lookup Tables in SS-based 2PC. Following the preprocessing model, the idea of using lookup tables (LUTs) in SS-based protocols was proposed by [47]. Their approach OTTT originally represents the entire circuit as one LUT leading to poor overall performance for non-trivial circuits. The idea of LUTs was also used by [30] to evalute AES S-boxes with malicious security. The concept of OTTT together with preprocessing from [30] was eventually combined in [33] for semi-honest security. In contrast to [47], its modification in [33] is also considered as replacement of sub-circuits to replace groups of gates by single multi-input gates. In addition, [33] proposes the current state-of-the-art LUT protocols for the semi-honest setting to replace their version of OTTT. One version, OP-LUT, is optimized for online communication while the other, SP-LUT, minimizes overall communication. We explain OP-LUT, SP-LUT and their version of OTTT in §3.1 and compare them to our protocol in §4. TinyTable [28] uses maliciously secure LUTs but no evaluation for multi-input gates is given. [33] notes that TinyTable suffers from low performance similar to their OTTT variant as it uses similar setup. [49] extends TinyTable to the multi-party case based on secret-sharing.

Several recent works use SP-LUT, e.g., [77] for comparisons in secure inference, [75] for evaluating math functions on floating-point numbers, and [76] for math functions on fixed-point numbers. The authors of [76] emphasize that LUTs play a crucial role in their protocol's performance. [75], [76] both also are of special interest for secure inference tasks.

Garbled LUTs. In Yao's GC setting [89], prior work noticed that 2-input/1-output gates can be extended into multi-input/multi-output gates to reduce the circuit evaluation overhead [45], [60], [73]. Fairplay [60] implemented Yao's GC protocols to evaluate gates with up to 3-input gates, but their method generalizes to an arbitrary number of inputs. The TASTY framework [45] implemented multi-input garbled gates including garbled-row reduction [72]. Recently, [73] proposed garbled circuits with multi-input/multi-output gates.

Boolean Circuit Compilers. Compilers that translate high-level code to binary circuits offer a higher level of abstraction when running MPC in the binary domain. Works in this area include CBMC-GC [36], HyCC [21], ObliVM [59], and LLVM-MPC [44]. Another approach is the deployment of existing hardware synthesis tools that take code in a hardware description language (HDL) such as Verilog as input. Examples of that include TinyGarble [83], TinyGMW [32], and Syncirc [69]. As demonstrated in [33], this approach can be extended to LUTs by utilizing and re-purposing LUT-based synthesis tools.

LUTNet for Neural Network Inference. Binary neural networks (BNN) are a promising approach to improving the efficiency of privacy-preserving machine learning (PPML) [79], [81], [90]. LUTNet [85] implements a LUT-based neural network architecture that yields significantly lower logic size than state-of-the-art BNNs and can be deployed on field-programmable gate arrays (FPGAs). This yields the potential of combining secure LUT protocols and LUTNet for efficient PPML as an option for future research.

6. Conclusion & Future Work

We presented FLUTE, a secure protocol for lookup table (LUT) evaluation that builds upon a vastly different idea than prior LUT approaches. We showed how this approach combines the best of two worlds that prior approaches for LUT evaluation reside in, namely an improved online communication and an improved overall communication.

We see three potential future directions. While we focus on using LUTs for efficient semi-honest 2PC as in [33], [75], [76], the first direction is to improve the security of our construction against malicious corruption. We anticipate that the general high-level design underlying our method, as mentioned in §3.2, could ease the transition. For instance, one may use either compilers like [15], [19], [40], [57] or adapt field-based protocols like [11] to enhance the security against malicious corruption. Another direction is the engineering of a novel compiler/toolchain that computes LUT circuits that are fine-tuned for our FLUTE technique, as the ones utilized in this work were generated with past approaches in mind. The third direction would be to investigate the impact of integrating our improved LUT evaluation approach with recent works on privacy-preserving machine learning [76] and floating point arithmetic [75], both of which significantly rely on prior LUT approaches. Our protocols could potentially replace some of their building blocks, leading to increased efficiency. However, a more interesting avenue for research would be to develop end-to-end applications such as SecFloat [75] using our enhanced LUT constructions. This would require a substantial engineering effort, as these applications use hybrid circuits with both binary and arithmetic sharing and in contrast to FLUTE, do not utilize function-dependent preprocessing.

Acknowledgments. This project received funding from the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230.

References

[1] "Berkeley Logic Synthesis. ABC: A System for Sequential Synthesis and Verification," https://github.com/berkeley-abc/abc, 2010, (visited on 11/25/2022).

[2] "Synopsys Inc. Design Compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, 2010, (visited on 11/25/2022).

[3] "Synopsys Inc. DesignWare Library - Datapath and Building Block IP," https://www.synopsys.com/dw/buildingblock.php, 2015, (visited on 11/25/2022).

[4] "Rust Programming Language v1.65.0," https://www.rust-lang.org, 2022, (visited on 11/25/2022).

[5] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A Survey on Homomorphic Encryption Schemes: Theory and Implementation," *ACM Comput. Surv.*, 2018.

[6] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," in *CCS*, 2013.

[7] S. Atapoor, N. P. Smart, and Y. T. Alaoui, "Private Liquidity Matching Using MPC," in *CT-RSA*, 2022.

[8] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *CRYPTO*, 1992.

[9] ——, "Precomputing Oblivious Transfer," in *CRYPTO*, 1995.

[10] ——, "Correlated Pseudorandomness and the Complexity of Private Computations," in *STOC*, 1996.

[11] A. Ben-Efraim, M. Nielsen, and E. Omri, "Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing," in *ACNS*, 2019.

[12] Y. Ben-Itzhak, H. Möllering, B. Pinkas, T. Schneider, A. Suresh, O. Tkachenko, S. Vargaftik, C. Weinert, H. Yalame, and A. Yanai, "ScionFL: Secure Quantized Aggregation for Federated Learning," *CoRR*, vol. abs/2210.07376, 2022, https://arxiv.org/abs/2210.07376.

[13] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference," in *ARES*, 2020.

[14] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical Secure Aggregation for Privacy-Preserving Machine Learning," in *CCS*, 2017.

[15] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs," in *CRYPTO*, 2019.

[16] J. Boyar and R. Peralta, "A Small Depth-16 Circuit for the AES S-Box," in *Information Security and Privacy Conference*, 2012.

[17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation," in *CCS*, 2019.

[18] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient Pseudorandom Correlation Generators: Silent OT Extension and More," in *CRYPTO*, 2019.

[19] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Secure Multiparty Computation with Sublinear Preprocessing," in *EUROCRYPT*, 2022.

[20] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame, "FLUTE: Fast and Secure Lookup Table Evaluations," in *IEEE S&P*, 2023.

[21] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation," in *CCS*, 2018.

[22] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning," *PETS*, 2020.

[23] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *CCSW@CCS*, 2019.

[24] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning," in *NDSS*, 2020.

[25] A. Choudhury and A. Patra, *Secure Multi-Party Computation Against Passive Adversaries*, 1st ed. Springer International Publishing, 2022.

[26] G. Couteau, P. Rindal, and S. Raghuraman, "Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes," in *CRYPTO*, 2021.

[27] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential Benchmarking Based on Multiparty Computation," in *FC*, 2016.

[28] I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci, "The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited," in *CRYPTO*, 2017.

[29] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption," in *CRYPTO*, 2012.

[30] I. Damgård and R. W. Zakarias, "Fast Oblivious AES A Dedicated Application of the MiniMac Protocol," in *AFRICACRYPT*, 2016.

[31] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, 2002.

[32] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni, "Automated Synthesis of Optimized Circuits for Secure Computation," in *CCS*, 2015.

[33] G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the Communication Barrier in Secure Computation using Lookup Tables," in *NDSS*, 2017.

[34] S. Even, O. Goldreich, and A. Lempel, "A Randomized Protocol for Signing Contracts," in *CRYPTO*, 1982.

[35] P. J. Fleming and J. J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Commun. ACM*, vol. 29, no. 3, 1986.

[36] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations," in *International Conference on Compiler Construction*, 2014.

[37] J. A. Garay, B. Schoenmakers, and J. Villegas, "Practical and Secure Solutions for Integer Comparison," in *PKC*, 2007.

[38] O. Goldreich, *The Foundations of Cryptography - Volume 2: Basic Applications*, 1st ed. Cambridge University Press, 2009.

[39] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *STOC*, 1987.

[40] V. Goyal, Y. Song, and C. Zhu, "Guaranteed Output Delivery Comes Free in Honest Majority MPC," in *CRYPTO*, 2020.

[41] D. Günther, M. Holz, B. Judkewitz, H. Möllering, B. Pinkas, T. Schneider, and A. Suresh, "Poster: Privacy-Preserving Epidemiological Modeling on Mobile Graphs," in *CCS*, 2022.

[42] A. Hamlin, N. Schear, E. Shen, M. Varia, S. Yakoubov, and A. Yerukhimovich, *Cryptography for Big Data Security*, 1st ed. Auerbach Publications, 2016.

[43] A. Hegde, H. Möllering, T. Schneider, and H. Yalame, "SoK: Efficient Privacy-preserving Clustering," *PETS*, vol. 2021.

[44] T. Heldmann, T. Schneider, O. Tkachenko, C. Weinert, and H. Yalame, "LLVM-Based Circuit Compilation for Practical Secure Computation," in *ACNS*, 2021.

[45] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for Automating Secure Two-Party Computations," in *CCS*, 2010.

[46] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO*, 2003.

[47] Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky, "On the Power of Correlated Randomness in Secure Computation," in *TCC*, 2013.

[48] M. H. S. Javadi, M. H. Yalame, and H. R. Mahdiani, "Small Constant Mean-Error Imprecise Adder/Multiplier for Efficient VLSI Implementation of MAC-Based Applications," *IEEE Trans. Computers*, 2020.

[49] M. Keller, E. Orsini, D. Rotaru, P. Scholl, E. Soria-Vazquez, and S. Vivek, "Faster Secure Multi-party Computation of AES and DES Using Lookup Tables," in *ACNS*, 2017.

[50] J. Kilian, "Founding Cryptography on Oblivious Transfer," in *STOC*, 1988.

[51] B. Knott, S. Venkataraman, A. Y. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure Multi-Party Computation Meets Machine Learning," in *NeurIPS*, 2021.

[52] V. Kolesnikov and R. Kumaresan, "Improved OT Extension for Transferring Short Secrets," in *CRYPTO*, 2013.

[53] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning," in *USENIX Security*, 2021.

[54] N. Koti, S. Patil, A. Patra, and A. Suresh, "MPClan: Protocol suite for privacy-conscious computations," 2022, https://ia.cr/2022/675.

[55] N. Koti, A. Patra, R. Rachuri, and A. Suresh, "Tetrad: Actively Secure 4PC for Secure Training and Inference," in *NDSS*, 2022.

[56] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, 1980.

[57] Y. Lindell and B. Pinkas, "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries," *Journal of Cryptology*, 2015.

[58] Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai, "Efficient Constant Round Multi-party Computation Combining BMR and SPDZ," in *CRYPTO*, 2015.

[59] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *IEEE S&P*, 2015.

[60] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - Secure Two-Party Computation System," in *USENIX Security*, 2004.

[61] F. Marx, T. Schneider, A. Suresh, T. Wehrle, C. Weinert, and H. Yalame, "HyFL: A Hybrid Approach For Private Federated Learning," *CoRR*, vol. abs/2302.09904, 2023, https://arxiv.org/abs/2302.09904.

[62] J. Münch, T. Schneider, and H. Yalame, "VASA: Vector AES Instructions for Security Applications," in *ACSAC*, 2021.

[63] M. Naor and B. Pinkas, "Oblivious Transfer and Polynomial Evaluation," in *STOC*, 1999.

[64] K. Y. Ngiam and I. W. Khor, "Big Data and Machine Learning Algorithms for Health-Care Delivery," *Lancet Oncol*, vol. 20, no. 5, 2019.

[65] T. D. Nguyen, P. Rieger, H. Chen, H. Yalame, H. Möllering, H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, S. Zeitouni, F. Koushanfar, A. Sadeghi, and T. Schneider, "FLAME: Taming Backdoors in Federated Learning," in *USENIX Security*, 2022.

[66] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *EUROCRYPT*, 1999.

[67] A. Patra, P. Sarkar, and A. Suresh, "Fast Actively Secure OT Extension for Short Secrets," in *NDSS*, 2017.

[68] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security*, 2021.

[69] ——, "SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation," in *IEEE HOST*, 2021.

[70] A. Patra and A. Suresh, "BLAZE: Blazing Fast Privacy-Preserving Machine Learning," in *NDSS*, 2020.

[71] L. R. Peter Rindal, "libOTe: An Efficient, Portable, and Easy to Use Oblivious Transfer Library," https://github.com/osu-crypto/libOTe, (visited on 11/30/2022).

[72] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure Two-Party Computation Is Practical," in *ASIACRYPT*, 2009.

[73] E. Pohle, A. Abidin, and B. Preneel, "Poster: Fast Evaluation of S-boxes in MPC," in *NDSS*, 2022.

[74] M. O. Rabin, "How To Exchange Secrets with Oblivious Transfer," Harvard University Technical Report TR-81, 1981, https://ia.cr/2005/187.

[75] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "SecFloat: Accurate Floating-Point meets Secure 2-Party Computation," in *IEEE S&P*, 2022.

[76] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "SiRnn: A Math Library for Secure RNN Inference," in *IEEE S&P*, 2021.

[77] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-Party Secure Inference," in *CCS*, 2020.

[78] D. Rathee, T. Schneider, and K. K. Shukla, "Improved Multiplication Triple Generation over Rings via RLWE-Based AHE," in *CANS*, 2019.

[79] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based Oblivious Deep Neural Network Inference," in *USENIX Security*, 2019.

[80] M. Rosulek and L. Roy, "Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits," in *CRYPTO*, 2021.

[81] M. Samragh, S. U. Hussain, X. Zhang, K. Huang, and F. Koushanfar, "On the Application of Binary Neural Networks in Oblivious Inference," in *CVPR*, 2021.

[82] T. Schneider and M. Zohner, "GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits," in *FC*, 2013.

[83] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits," in *IEEE S&P*, 2015.

[84] A. Treiber, D. Müllmann, T. Schneider, and I. S. genannt Döhmann, "Data Protection Law and Multi-Party Computation: Applications to Information Exchange between Law Enforcement Agencies," in *WPES*, 2022.

[85] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference," *IEEE Trans. Computers*, 2020.

[86] X. Wang, S. Ranellucci, and J. Katz, "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation," in *CCS*, 2017.

[87] C. Wolf, J. Glaser, and J. Kepler, "Yosys - A Free Verilog Synthesis Suite," in *Austrian Workshop on Microelectronics*, 2013.

[88] A. C.-C. Yao, "Protocols for Secure Computations (Extended Abstract)," in *FOCS*, 1982.

[89] ——, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS*, 1986.

[90] W. Zhu, M. Wei, X. Li, and Q. Li, "SecureBiNN: 3-Party Secure Computation for Binarized Neural Network Inference," in *ESORICS*, 2022.

Appendix

1. Setup Phase in FLUTE

Recall from §3.2.2 that the goal of $\mathcal{F}_{\mathsf{ANDM}}^{\mathsf{pre}}$ is to generate $[\lambda_{\mathcal{Q}}]$, given the $\langle\cdot\rangle$-shares of δ elements $\mathcal{I} = \{x_1, \ldots, x_\delta\}$, for all $\mathcal{Q} \in 2^{\mathcal{I}}$. In FLUTE, we implement $\mathcal{F}_{\mathsf{ANDM}}^{\mathsf{pre}}$ using the $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}$ functionality as the basis. Given the $[\cdot]$-shares of two bits $u, v \in \{0,1\}$, $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}$ computes $[uv]$ and we instantiate the same using Beaver's multiplication method [8]. The formal protocol $\Pi_{\mathsf{AND}}^{\mathsf{pre}}$ that realises $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}$ in the 2PC semi-honest setting is given in Fig. 11. Protocol $\Pi_{\mathsf{AND}}^{\mathsf{pre}}$ has a communication of $4 + |\mathsf{MT}|$ bits, where MT denotes a boolean multiplication triple, and is realised using the $\mathcal{F}_{\mathsf{genMT}}$ functionality (cf. §B.1) in this work.

Deriving a protocol $\Pi_{\mathsf{ANDM}}^{\mathsf{pre}}$ that implements $\mathcal{F}_{\mathsf{ANDM}}^{\mathsf{pre}}$ from $\Pi_{\mathsf{AND}}^{\mathsf{pre}}$ now is straightforward. Recall that the goal is to compute $[\lambda_{\mathcal{Q}}]$ for all $\mathcal{Q} \in 2^{\{x_1,\ldots,x_k\}}$ given inputs $[\lambda_{x_1}], ..., [\lambda_{x_k}]$. In the first step, we run $\Pi_{\mathsf{AND}}^{\mathsf{pre}}$ for each $\mathcal{Q}$ where $|\mathcal{Q}| = 2$. From the results, the next step then runs $\Pi_{\mathsf{AND}}^{\mathsf{pre}}$ for each $\mathcal{Q}$ where $|\mathcal{Q}| \in \{3,4\}$ utilizing the previously computed values. For instance, $[\lambda_{x_1x_2x_3}]$ can be computed from $[\lambda_{x_1x_2}], [\lambda_{x_3}]$ and $[\lambda_{x_1x_2x_3x_4}]$ can be computed from $[\lambda_{x_1x_2}], [\lambda_{x_3x_4}]$. By continuing this scheme which doubles the size of considered $\mathcal{Q}$ in each step, eventually all $\mathcal{Q} \in 2^{\{x_1,\ldots,x_k\}}$ are covered. As for each $\mathcal{Q}$ with $|\mathcal{Q}| \geq 2$ one multiplication is required, the total communication amounts to $(2^k - k - 1)(4 + |\mathsf{MT}|)$.

Protocol $\Pi_{\mathsf{AND}}^{\mathsf{pre}}([u],[v])$

Input: $[\cdot]$-shares of $u, v \in \{0,1\}$.
Output: $[\cdot]$-shares of $z = uv$.

Computation:

1. Servers invoke the $\mathcal{F}_{\mathsf{genMT}}$ functionality to generate the $[\cdot]$-shares of a boolean multiplication triple MT, i.e., $([a],[b],[c])$ with $c = a \wedge b$.
2. Server S_i, for $i \in \{0,1\}$, computes $[d]_i = [u]_i \oplus [a]_i, [e]_i = [v]_i \oplus [b]_i$ and sends $([d]_i, [e]_i)$ to S_{1-i}.
3. Servers reconstruct $d = [d]_0 \oplus [d]_1$ and $e = [e]_0 \oplus [e]_1$.
4. S_i, for $i \in \{0,1\}$, sets $[z]_i = i \cdot de \oplus d[b]_i \oplus e[a]_i \oplus [c]_i$.

Figure 11: Instantiating the $\mathcal{F}_{\mathsf{AND}}^{\mathsf{pre}}$ functionality in the 2PC semi-honest setting using Beaver's multiplication method from [8].

2. Silent OT for Multiplication Triples, OP-LUT and SP-LUT

In recent years, the communication required for rOT_ℓ^m has been significantly decreased by silent OT extension introduced by [18]. The state-of-the-art silent OT extension [17], [26] that is proven secure under variants of the learning parity with noise (LPN) assumption can decrease the cost of each random OT instance rOT_ℓ^1 to less than a single bit. In this section, we elaborate on using silent OT for generating multiplication triples that are required by FLUTE and OTTT (§B.1), and used to improve the communication for OP-LUT and SP-LUT (§B.2).

2.1. Generation of Multiplication Triples. A boolean multiplication triple $[x],[y],[z]$ where $z = x \wedge y$ can be obtained from two random OTs using no further communication [6]. Due to low random OT cost from silent OT, we use this technique to generate multiplication triples which yields $|\mathsf{MT}| = |\mathsf{rOT}_1^2|$. For the sake of completeness, we give the complete protocol in Fig. 12.

Protocol $\Pi_{\mathsf{genMT}}()$

Output: $[\cdot]$-shares of independently and randomly sampled $x, y \in \mathbb{Z}_2$, and $z = x \wedge y$.

1. S_0, S_1 call rOT_1^1 twice, once with S_0 being the sender and once with S_1 being the sender.
   - S_0 as sender: S_0 receives random $a_0, a_1 \in \{0,1\}$, S_1 receives random $r \in \{0,1\}$ as well as a_r.
   - S_1 as sender: S_1 receives random $b_0, b_1 \in \{0,1\}$, S_0 receives random $s \in \{0,1\}$ as well as b_s.
2. S_0, S_1 define their outputs as:
   - S_0: $x^0 = s, y^0 = a_0 \oplus a_1, z^0 = sy^0 \oplus b_s \oplus a_0$.
   - S_1: $x^1 = r, y^1 = b_0 \oplus b_1, z^1 = ry^1 \oplus a_r \oplus b_0$.

Figure 12: Instantiating the $\mathcal{F}_{\mathsf{genMT}}$ functionality in the 2PC semi-honest setting for boolean multiplication triple generation from [6].

2.2. OT Conversions for Optimizing OP-LUT and SP-LUT. While [33] uses their own $\binom{2^\delta}{1}$-OT_ℓ^m and $\binom{2^\delta}{1}$-rOT_ℓ^m protocols to implement OP-LUT and SP-LUT, silent OT can significantly decrease the cost of these LUT approaches. As SilentOT implements rOT_ℓ^m, additional OT conversions become necessary.

First, in the 2PC semi-honest setting, $\binom{2^\delta}{1}$-rOT_ℓ^m as required for SP-LUT can be reduced to $\mathsf{rOT}_\ell^{m\delta}$ without further communication using the construction from [63]. Thus, it holds that $|\binom{2^\delta}{1}\text{-}\mathsf{rOT}_\ell^m| = |\mathsf{rOT}_\ell^{m\delta}|$ for the 2PC semi-honest setting considered in this work. The OT instantiation for a single SP-LUT can hence be replaced by $\mathsf{rOT}_\sigma^\delta$ using silent OT.

Furthermore, such 1-out-of-N random OT can be used to obtain 1-out-of-N actual OT as required for OP-LUT. The main idea is from [9] which allows to reduce $\binom{2}{1}$-OT_ℓ^m to rOT_ℓ^m using an additional communication of $m(1+2\ell)$ bits.[4] Here, the sender's outputs from the random OT are used as one-time pad to encrypt its inputs to the actual OT which are then sent resulting in 2ℓ bits of additional communication per instance. The remaining bit is used beforehand to correct the random choice bit that the receiver got from the random OT to its actual choice bit. It is easy to see that this approach generalizes to obtaining $\binom{2^\delta}{1}$-OT_ℓ^m from $\binom{2^\delta}{1}$-rOT_ℓ^m plus $m(\delta + 2^\delta\ell)$ bits of communication. Thus, it holds that $|\binom{2^\delta}{1}\text{-}\mathsf{OT}_\ell^m| = |\binom{2^\delta}{1}\text{-}\mathsf{rOT}_\ell^m| + m(\delta + 2^\delta\ell) = |\mathsf{rOT}_\ell^{m\delta}| + m(\delta + 2^\delta\ell)$ bits. Recall that the communication for $\binom{2^\delta}{1}$-OT_ℓ^m in [33] is reduced by δ bits by keeping the receivers choice string random (cf., §3.1.2). This is equiva-

4. In [9], only $\ell = 1$ is considered, but this easily generalizes to arbitrary ℓ using techniques from [10].

lent to not derandomizing the choice string in the aforementioned conversion decreasing the additional communication from $m(\delta + 2^\delta \ell)$ bits down to $m2^\delta \ell$ bits. Thus, the OT instantiation for a single OP-LUT using that optimization can be replaced by $\mathsf{rOT}^{\delta}_{2^\delta \sigma}$ using silent OT and $2^\delta \cdot 2^\delta \sigma$ bits additional communication.

3. Optimizations Using an Untrusted Helper

In this section, we discuss how FLUTE and prior LUT protocols can perform better with an additional helper server during the setup phase. In detail, we augment the semi-honest 2PC setting by a semi-honest helper server $S_\mathcal{H}$ that does not collude with S_0 or S_1 and does not send or receive any messages in the protocols' online phases. This is motivated by the overall communication of OTTT, OP-LUT and FLUTE being dominated by the setup phase. As SP-LUT, especially when utilizing silent OT, pushes almost all communication to the online phase, we see no promising approach of improving its setup communication using a helper and hence omit it.

For our optimizations, we assume a pre-shared key setup among the 2PC servers S_0, S_1 and the helper $S_\mathcal{H}$, where $S_i, S_\mathcal{H}$ share a PRF key $\mathcal{K}_{i,\mathcal{H}}$, for $i \in \{0,1\}$. This allows $S_\mathcal{H}$ to generate the same randomness used by the other servers without interaction. Our setting with a helper resembles to the three party semi-honest setting in ASTRA [23] as our 2PC baseline ABY2.0 [68] and ASTRA share similar sharing semantics. However, we consider the helper to be available only during the setup phase and analyzing our approach in a three-party honest majority setting is left for future work.

3.1. OTTT and OP-LUT. Recall that OTTT and OP-LUT use the exact same approach and only differ in how they compute their sharings of the randomly rotated LUT (cf. §3.1.1 and §3.1.2). We observe that this setup can be replaced by yet another approach when using an additional helper which is significantly more efficient than both OTTT and OP-LUT. Using their pre-shared keys, $S_0, S_\mathcal{H}$ sample random r and $S_1, S_\mathcal{H}$ sample random s to obtain a sharing of a random LUT rotation $\theta = r \oplus s$ as it is used in OTTT and OP-LUT. Then, $S_\mathcal{H}$ locally computes the LUT rotated by θ. Note that this does not violate the protocol's security because while $S_\mathcal{H}$ knows the rotation and resulting rotated LUT, it never sees any actual inputs and does not collude with one of the other parties. The straightforward next step would be to let $S_\mathcal{H}$ set up two shares $\mathsf{T}^0, \mathsf{T}^1$ (cf. §3.1.1) of the rotated LUT and sending them to S_0, S_1 resulting in $2 \cdot 2^\delta \sigma$ setup communication. Instead, we let $S_0, S_\mathcal{H}$ use $\mathcal{K}_{0,\mathcal{H}}$ to randomly sample T^0 and only send T^1. Thus, the overall setup communication is only $2^\delta \sigma$ bits and this clearly outperforms OP-LUT and OTTT. Even when ignoring the cost of required random OTs for OP-LUT, we thus outperform its setup communication by a factor of 2^δ. For OTTT, the factor when ignoring the cost of generating multiplication triples still is $4\delta - 4$.

3.2. FLUTE. Regarding FLUTE, recall that the setup communication exclusively comes from computing the AND of values $[\lambda_x]$ and $[\lambda_y]$. For each AND, one multiplication triple and an additional 4 bits of communication is required. We show that using a helper $S_\mathcal{H}$, the cost per AND can be decreased to a single bit yielding an improvement of over $4\times$.

The first essential change is that for each secret-shared value v that is used at some point of the computation, $S_\mathcal{H}$ also knows λ_v^0, λ_v^1 that are already selected in the setup phase. Each time that S_0, S_1 would sample these values for some wire v, this can be achieved by $S_i, S_\mathcal{H}$ for $i \in \{0,1\}$ instead using $\mathcal{K}_{i,\mathcal{H}}$ to generate λ_v^i. Now, assume that the AND of values $[\lambda_x]$ and $[\lambda_y]$ is to be computed. As $S_\mathcal{H}$ knows all shares, it can locally compute $\lambda_x \wedge \lambda_y$. Then, $S_0, S_\mathcal{H}$ sample random λ_{xy}^0 using $\mathcal{K}_{0,\mathcal{H}}$, and $S_\mathcal{H}$ locally computes $\lambda_{xy}^1 = \lambda_x \wedge \lambda_y \oplus \lambda_{xy}^0$ and sends it to S_1. Thus, the cost of each such multiplication is decreased from $4 + |\mathsf{MT}|$ bits down to 1 bit. The security of the resulting protocol changes directly follows from the security of ASTRA [23].

4. Security Proof

In this section, we discuss the security details of FLUTE. Since we use the 2PC protocol of ABY2.0 [68] as our baseline without any modifications, we inherit the security of ABY2.0. Hence, we focus on the security of the multi-fan-in inner product protocol Π_{IP} (cf. §3.2.1) proposed in this paper. Furthermore, since the Π_{LUT} protocol for LUT evaluation (Fig. 6) is an optimized variant of Π_{IP}, we then briefly elaborate on how the security proof for Π_{IP} translates to Π_{LUT}.

$\mathcal{F}_{\mathsf{IP}}$ interacts with $\{S_0, S_1\}$ and the adversary $\mathcal{A}$.

Input: $\mathcal{F}_{\mathsf{IP}}$ receives $\langle\cdot\rangle$-shares for β boolean vectors, denoted by $(\langle \vec{\mathbf{x}}^1 \rangle, \ldots, \langle \vec{\mathbf{x}}^\beta \rangle)$, from the respective servers in $\{S_0, S_1\}$, with the vectors having d elements each.

Computation: $\mathcal{F}_{\mathsf{IP}}$ reconstructs the vectors from its $\langle\cdot\rangle$-shares and computes

$$z = \vec{\mathbf{x}}^1 \odot \cdots \odot \vec{\mathbf{x}}^\beta = \bigoplus_{j=1}^{\mathsf{d}} \Big(\bigwedge_{i=1}^{\beta} \vec{\mathbf{x}}^i_j \Big).$$

$\mathcal{F}_{\mathsf{IP}}$ then samples random $\lambda_z^0, \lambda_z^1 \in \{0,1\}$ and sets $\mathsf{m}_z = z \oplus \lambda_z^0 \oplus \lambda_z^1$.

Output: $\mathcal{F}_{\mathsf{IP}}$ sends $(\mathsf{m}_z, \lambda_z^i)$ to S_i, for $i \in \{0,1\}$.

Figure 13: Ideal functionality $\mathcal{F}_{\mathsf{IP}}$ for Multi-Fan-In Inner Product in the 2PC semi-honest setting.

The ideal functionality for the Π_{IP} protocol, denoted by $\mathcal{F}_{\mathsf{IP}}$, is given in Fig. 13 and we prove security using the standard real world / ideal world paradigm. We provide the simulation for the case of a corrupt S_0. Since the protocol is symmetric, the simulation for the case of corrupt S_1 follows similarly. Our proof works in the $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$-hybrid model and the security of $\Pi^{\mathsf{pre}}_{\mathsf{ANDM}}$ implementing $\mathcal{F}^{\mathsf{pre}}_{\mathsf{ANDM}}$ reduces to that of $\Pi^{\mathsf{pre}}_{\mathsf{AND}}$ implementing $\mathcal{F}^{\mathsf{pre}}_{\mathsf{AND}}$ as discussed in §A.

Theorem 1. In the $\{\mathcal{F}_{\text{key}}, \mathcal{F}^{\text{pre}}_{\text{ANDM}}\}$-hybrid model, Π_{IP} (cf. Fig. 5) securely realizes the functionality $\mathcal{F}_{\text{IP}}$ against a semi-honest adversary $\mathcal{A}$, who corrupts S_0.

Proof. Let $\mathcal{A}$ denote the semi-honest adversary that corrupts S_0 during the protocol Π_{IP}. We now present the steps of the ideal-world adversary (simulator) $\mathcal{S}_{\text{IP}}$ for $\mathcal{A}$ for this case. Note that the Π_{IP} protocol is simply one component of the underlying 2PC protocol ABY2.0 [68], which includes other stages such as input sharing and output reconstruction, as discussed in §2.4. As a result, we presume that all the stages till Π_{IP} are correctly simulated, and we use the information obtained by $\mathcal{S}_{\text{IP}}$ during those stages in this simulation. For instance, as discussed in ABY2.0 [68], the simulation of the shared-key setup $\mathcal{F}_{\text{key}}$ enables $\mathcal{S}_{\text{IP}}$ to receive the PRF key that the adversary $\mathcal{A}$ (in other words, the corrupt S_0) uses in the protocol and hence $\mathcal{S}_{\text{IP}}$ learns all the intermediate values of the circuit in the clear. Similarly, during the simulation for input sharing, $\mathcal{S}_{\text{IP}}$ has to simulate nothing for the inputs of S_0 since $\mathcal{A}$ is not receiving any messages in this case. Instead, $\mathcal{S}_{\text{IP}}$ receives the m_v values from $\mathcal{A}$ on behalf of S_1. For the case of S_1's inputs, $\mathcal{S}_{\text{IP}}$ executes the protocol steps honestly, assuming the inputs of S_1 to be all 0.

With respect to the multi-fan-in inner product protocol, the setup phase involves locally sampling random masks $\lambda^0_z, \lambda^1_z \in \{0, 1\}$ and the invocation of the ideal functionality $\mathcal{F}^{\text{pre}}_{\text{ANDM}}$ which generates the required correlated randomness. Since we make only black-box access to $\mathcal{F}^{\text{pre}}_{\text{ANDM}}$, the simulation for the same follows from the security of the underlying primitive used to instantiate $\mathcal{F}^{\text{pre}}_{\text{ANDM}}$ (cf. §A). During the online phase, $\mathcal{S}_{\text{IP}}$ follows the step honestly using the data obtained from the corresponding setup phase. The resulting view is indistinguishable from a real protocol execution for the following reason: The only message that S_0 receives from S_1 is $[v]_1 \in \{0, 1\}$. In both the simulation and the real protocol execution, this value is obtained by taking the XOR of an intermediate value and a random mask λ^1_z that is chosen by S_1 and unknown to S_0. Note that this mask is not used again to mask any other value. Therefore, to S_0, $[v]_1$ appears as a uniformly random bit in both the real protocol execution and the simulation. $\square$

The security of Π_{LUT} follows similar to Π_{IP}, but with some minor modifications in the prior simulation of Π_{IP}, as listed below:

1. For the setup phase, the simulator randomly samples σ values $\lambda^0_{\vec{\mathbf{z}}_i}, \lambda^1_{\vec{\mathbf{z}}_i} \in \{0, 1\}$ for $1 \le i \le \sigma$ instead of a single value; one for each output wire.
2. The simulation requires only a single invocation of the $\mathcal{F}^{\text{pre}}_{\text{ANDM}}$ ideal functionality as opposed to d many in Π_{IP}.
3. The simulator sends a vector of σ bits $[\vec{\mathrm{v}}]_1$ to S_0 on behalf of S_1, instead of a single bit.

Note that the simulation of the setup phase remains independent of the number of output wires, denoted by σ. This does not compromise the security of FLUTE because a single instance of $\mathcal{F}^{\text{pre}}_{\text{ANDM}}$ provides all the necessary values for local computation in the online phase. Moreover, each message sent in the online phase is concealed by a unique and random one-time mask, denoted by $\lambda^0_{\vec{\mathbf{z}}_i}$ or $\lambda^1_{\vec{\mathbf{z}}_i}$, ensuring the security of FLUTE. Furthermore, using different and independent random masks guarantees that the shares of different output wires belonging to one server are not correlated.

5. Additional Benchmark Results

Here, we give complementary results from our theoretical and practical evaluation and benchmarks. Regarding our theoretical considerations per LUT from §4.1, Fig. 14 compares the online complexity of OTTT/OP-LUT, SP-LUT from [33] as well as FLUTE. Recall that the use of silent OT does not affect the online phase.

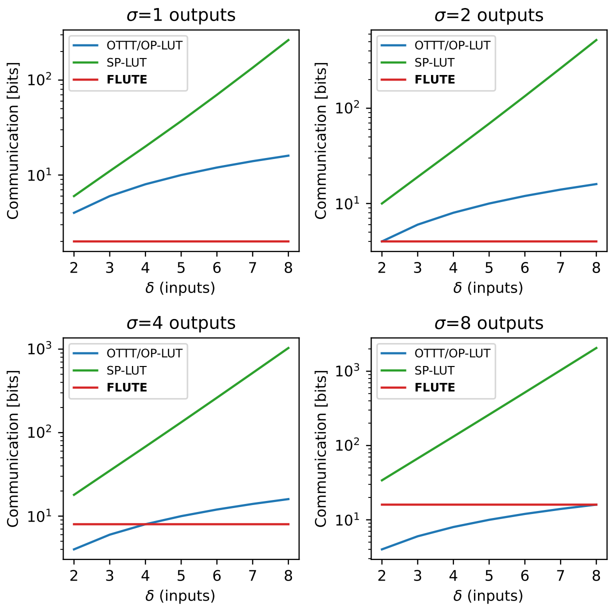


Figure 14: Online communication for different LUT sizes with $2 \le \delta \le 8$ inputs and $\sigma \in \{1, 2, 4, 8\}$ outputs.

Table 7 gives the online improvement factor of FLUTE over SP-LUT. Table 8 provides a full analytical communication comparison of all considered circuits when using Yao, OTTT$^+$, SP-LUT$^+$, FLUTE as well as the helper variants OTTT$^\mathcal{H}$ and FLUTE$^\mathcal{H}$. Table 9 contains our implementation's communication and run time for LAN and WAN settings when using ABY2.0$^+$ or FLUTE on different circuits.

TABLE 7: Improvement factor of online communication of FLUTE over SP-LUT when evaluating one δ-input σ-output LUT.

δ \ σ	1	2	3	4	5	6	7	8
2	3.0	2.5	2.3	2.3	2.2	2.2	2.1	2.1
3	5.5	4.8	4.5	4.4	4.3	4.3	4.2	4.2
4	10.0	9.0	8.7	8.5	8.4	8.3	8.3	8.3
5	18.5	17.3	16.8	16.6	16.5	16.4	16.4	16.3
6	35.0	33.5	33.0	32.8	32.6	32.5	32.4	32.4
7	67.5	65.8	65.2	64.9	64.7	64.6	64.5	64.4
8	132.0	130.0	129.3	129.0	128.8	128.7	128.6	128.5

TABLE 8: Analytical comparison of total communication in KiB (first online, then total) of Yao's garbled circuits [89] using three-halves garbling [80], OTTT [30], [47] with silent OT [17] (OTTT$^+$), SP-LUT [33] with silent OT [17] (SP-LUT$^+$) and FLUTE. The case of OTTT and FLUTE with a helper server is marked with a superscript "$^\mathcal{H}$". We do not consider the input phase leading to zero online communication for Yao.

Circuit	Yao		OTTT$^+$		SP-LUT$^+$		FLUTE		OTTT$^\mathcal{H}$		FLUTE$^\mathcal{H}$	
Basic Functionalities												
Add-RC	0	0.73	0.028	46.68	1.598	1.60	0.014	1.22	0.028	1.61	0.014	0.30
Sub-RC	0	0.75	0.026	45.61	1.557	1.56	0.014	1.19	0.026	1.57	0.014	0.29
Mul-RC	0	93.59	1.258	1,577.94	54.961	55.04	0.514	48.60	1.258	55.59	0.514	11.86
Add-LF	0	3.73	0.049	54.50	1.900	1.90	0.017	2.09	0.049	1.92	0.017	0.51
Sub-LF	0	3.73	0.049	54.50	1.900	1.90	0.017	2.09	0.049	1.92	0.017	0.51
Mul-LF	0	96.52	0.917	1,405.54	48.910	48.96	0.452	34.29	0.917	49.37	0.452	8.44
GT-Tree	0	2.09	0.020	15.26	0.528	0.53	0.005	1.22	0.020	0.54	0.005	0.29
AES S-box	0	0.80	0.002	7.42	0.251	0.25	0.002	0.13	0.002	0.25	0.002	0.03
Floating-Point Functionalities												
FP-Add	0	51.09	0.622	567.08	19.924	19.96	0.186	23.28	0.622	20.24	0.186	5.64
FP-Sub	0	49.27	0.595	539.33	18.968	19.00	0.179	21.92	0.595	19.27	0.179	5.31
FP-Mul	0	106.01	1.263	1,083.16	37.900	37.97	0.340	48.74	1.263	38.53	0.340	11.76
FP-Div	0	128.16	1.735	1,543.49	54.613	54.72	0.534	69.51	1.735	55.48	0.534	16.82
FP-Sqr	0	49.03	0.534	647.98	22.420	22.45	0.194	21.39	0.534	22.69	0.194	5.20
FP-Sqrt	0	70.90	0.893	753.65	26.770	26.82	0.268	35.92	0.893	27.22	0.268	8.68
FP-Sin	0	122.23	1.536	1,459.79	50.996	51.09	0.461	57.59	1.536	51.76	0.461	13.95
FP-Cos	0	122.51	1.498	1,444.20	50.495	50.58	0.462	57.15	1.498	51.24	0.462	13.84

TABLE 9: Actual communication in KiB and run time for LAN and WAN in ms (first online, then total) for FLUTE and ABY2.0 [68] with silent OT [17] (ABY2.0$^+$).

Circuit	Communication [KiB]				Run time for LAN [ms]				Run time for WAN [ms]			
	ABY2.0$^+$		FLUTE		ABY2.0$^+$		FLUTE		ABY2.0$^+$		FLUTE	
Basic Functionalities												
Add-RC	–	–	0.318	1.80	–	–	13	32	–	–	415	827
Sub-RC	–	–	0.318	1.76	–	–	13	32	–	–	422	846
Mul-RC	–	–	1.910	50.30	–	–	116	444	–	–	2,084	2,815
Add-LF	0.207	0.32	0.119	2.45	7	11	6	33	207	307	114	539
Sub-LF	0.207	0.32	0.119	2.45	7	11	6	33	213	317	110	549
Mul-LF	1.497	2.79	0.723	34.86	48	89	38	290	1,321	1,570	450	1,112
GT-Tree	0.299	0.39	0.107	1.59	7	10	4	22	298	404	109	534
AES S-box	–	–	0.035	0.42	–	–	1	12	–	–	5	413
Floating-Point Functionalities												
FP-Add	2.979	4.10	0.798	24.19	107	138	46	205	3,599	3,730	945	1,484
FP-Sub	2.946	4.07	0.863	22.91	105	135	47	200	3,595	3,728	935	1,500
FP-Mul	3.191	5.55	0.936	49.64	109	170	58	347	3,078	3,237	864	1,565
FP-Div	11.855	14.73	3.214	72.49	435	506	187	585	15,611	15,781	4,088	4,942
FP-Sqr	1.777	2.88	0.591	22.08	62	93	33	186	1,865	1,996	555	1,136
FP-Sqrt	8.091	9.70	2.114	38.06	290	336	120	342	11,041	11,187	2,813	3,442
FP-Sin	4.458	7.20	1.304	58.75	155	222	91	454	4,731	4,900	1,286	2,077
FP-Cos	4.567	7.32	1.303	58.30	158	226	88	448	4,902	5,075	1,292	2,133