MAESTRO: Multi-party AES using Lookup Tables

Hiraku Morita
Aarhus University
Aarhus, Denmark
University of Copenhagen
Copenhagen, Denmark
hiraku@cs.au.dk

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

Kunihiko Sadakane
The University of Tokyo
Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp

Peter Scholl
Aarhus University
Aarhus, Denmark
peter.scholl@cs.au.dk

Kazunari Tozawa
The University of Tokyo
Tokyo, Japan
tozawa.kazunari@mail.u-tokyo.ac.jp

Daniel Tschudi
Concordium
Zürich, Switzerland
dt@concordium.com

ABSTRACT

Secure multi-party computation (MPC) enables multiple distrusting parties to jointly compute a function while keeping their inputs private. Computing the AES block cipher in MPC, where the key and/or the input are secret-shared among the parties is important for various applications, particularly threshold cryptography.

In this work, we propose a family of dedicated, high-performance MPC protocols to compute the non-linear S-box part of AES in the honest majority setting. Our protocols come in both semi-honest and maliciously secure variants. The core technique is a combination of lookup table protocols based on random one-hot vectors and the decomposition of finite field inversion in $GF(2^8)$ into multiplications and inversion in the smaller field $GF(2^4)$, taking inspiration from ideas used for hardware implementations of AES. We also apply and improve the analysis of a batch verification technique for checking inner products with logarithmic communication. This allows us to obtain malicious security with almost no communication overhead, and we use it to obtain new, secure table lookup protocols with only $O(\sqrt{N})$ communication for a table of size N, which may be useful in other applications.

Our protocols have different trade-offs, such as having a similar round complexity as previous state-of-the-art but 37% lower bandwidth costs, or having 27% fewer rounds and 16% lower bandwidth costs. An experimental evaluation in various network conditions using three party replicated secret sharing shows improvements in throughput between 23% and 27% in the semi-honest setting. For malicious security, we improve throughput by 46% and 270% in LAN and by up to 453% in WAN due to a new multiplication verification protocol.

1 INTRODUCTION

Secure multi-party computation (MPC) has become a practical component to realize privacy-preserving computation and to improve both privacy and security of existing processes and data flows. Using MPC, a set of distrusting parties can jointly evaluate a function while keeping their own input private. The security of this approach relies on distributed trust that no adversary corrupts more parties than the allowed corruption threshold.

Often, MPC protocols are designed to support generic computation with good performance over commonly used functions. For specific, complex functions, such as symmetric-key primitives like block ciphers, using a generic MPC protocol will not yield optimal performance. In these cases, the effort of designing highly-specialized and high performance MPC protocols for essential building blocks is worthwhile to improve the overall performance of privacy-preserving systems and applications.

An MPC protocol to evaluate the AES block cipher with secret-shared key and input has various applications that all benefit from improved performance of the primitive in MPC. Such specialized protocols for AES will be the main focus of this work.

AES evaluations in MPC can be used when clients communicate and exchange data with a cluster of MPC engines in a secure and opaque manner using, e.g., oblivious TLS [2] or clients secret-share a secret key to enable MPC parties to distributively decrypt [10, 50] and arbitrarily process their data on their behalf [48] in a secure way, e.g., for secure IoT data collection and processing [1]. Furthermore, it enables secure database joins [43], keyword search [30], private set intersection [36] or allows to increase the trust in systems that rely on centralized secrets, like the Key Distribution Center in the Kerberos authentication protocol or brokered identifcation systems [17]. Distributed variants [6] and similar distributed authentication protocols [9] rely on AES evaluations with a secret-shared key or can use AES as an oblivious PRF with high confidence in its security. The general case of this class of applications is *threshold cryptography*, which uses MPC to protect cryptographic keys while they are used, adding a layer of distributed trust to a secure system. NIST identified this use-case and initiated the multi-party threshold cryptography project[1] to study, among others, symmetric-key functions like AES-based enciphering, CMAC and HMAC in a *threshold* way.

Since cryptographic primitives tend to be fairly complex to evaluate inside an MPC protocol, much effort has been put into designing *MPC-friendly* variants of standard primitives such as hash functions [3], block ciphers [4] and pseudorandom functions [34]. This approach comes with two major drawbacks, however. Firstly, since these primitives are relatively new, they have not been subjected to the same level of scrutiny from cryptanalysts as established, longstanding primitives like AES or SHA-256, so there is less confidence in their security. Secondly, none of these primitives are standardized or even in widespread use. This rules out deploying these types of constructions in applications where MPC must

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

be integrated into an existing system, which only uses standard cryptographic primitives. Aside from these adoption and integration challenges, some MPC-friendly primitives also require many more rounds, e.g., MiMC [3] needs ≈ 8 times more rounds for the same security level compared to AES. Furthermore, large (prime) field arithmetic or extensive use of bit-bit multiplication (e.g., in the linear layers of LowMC [4]) makes them relatively slow to evaluate in plain software.

1.1 Contribution

We present a family of MPC protocols to evaluate the AES block cipher in the multi-party, honest majority setting with both semi-honest and maliciously secure variants.

Our contribution is three-fold:

- We securely compute the AES block cipher with novel protocols to compute the inversion in $GF(2^8)$ (where 0 maps to 0). The proposed protocols either rely on lookup table protocols based on preprocessed random one-hot vectors, or on the isomorphism between $GF(2^8)$ and $GF((2^4)^2)$ or on a combination of both techniques. This results in a range of different trade-offs in computational complexity, bandwidth costs and round complexity. Malicious security (with abort) is achieved by a multiplication verification check with logarithmic communication. The overall design focus is maximizing online phase and total throughput.
- We present several lookup table protocols for (t, n) replicated and (n, n) additive secret-sharing, which may be of independent interest. In particular, our protocol based on replicated secret sharing is maliciously secure and only needs $O(\sqrt{N})$ communication for a table of size N. All prior malicious protocols based on information-theoretic primitives require $\Omega(N)$ communication.
- We implemented some of the described protocols in the 3-party setting, and experimentally verify their performance in multiple network settings. We obtain improvements of 27% and 23% for online phase and total throughput in the semi-honest setting, compared with the best prior work of Chida et al. [21]. For malicious security, we improve the total throughput from 46% to 270% in LAN while decreasing communication by up to 93%. In WAN, the total throughput is increased up to 453%.

1.2 Technical Overview

We now give a brief overview of our main protocols. Their performance characteristics in the 3-party setting, together with those of the most competitive related work, are summarised in Table 1.

Here, the communication volume represents the number of bits sent by each party, excluding the cost of key expansion steps. As shown in Table 1, the execution of a 10-round oblivious AES protocol involves a communication volume of 4320 bits in 22 communication rounds. We have reduced the round complexity by 27% and communication by 16% compared to the state-of-the-art [21], whilst also adding malicious security with almost no communication overhead.

S-box via $GF(2^4)$ inversion. Our main approach to evaluating the S-box in MPC views the inversion in $GF(2^8)$ as an extension of $GF(2^4)$ at the cost of one inversion and 3 multiplications in $GF(2^4)$. We observe that this representation considerably simplifies the complexity of the S-box and identify suitable MPC subprotocols for the $GF(2^4)$ inversion. Although methods for finite field inversion via a tower of field extensions are well-known [39] and have been applied to AES [12, 20], as far as we are aware, they have not been explicitly considered in an MPC context.

$GF(2^4)$-circuit: inversion as an arithmetic circuit. First, we consider the simple approach of computing $x^{-1} = x^{14} = x^2 \cdot x^4 \cdot x^8$ in $GF(2^4)$. This can be done with just 2 multiplications, since squaring is $GF(2)$-linear. This already reduces the communication by more than one third, compared with a $GF(2^8)$ circuit-based approach [21], while preserving the same round complexity.

$GF(2^4)$-LUT-16: inversion with a lookup table. Our next variant uses lookup table techniques to evaluate the inverse. Here, we adapt techniques from the dishonest majority MPC literature [18, 25, 40], which allow to offload the work of computing a lookup table to a preprocessing phase. The high-level idea is to use the preprocessing phase to compute a secret-shared, random one-hot binary vector (that is, all-zero except for a single position) of length equal to the table size, which we do based on a protocol from [40, 42]. Using this, in the online phase the parties can open the masked input and then compute the table lookup with a single linear combination. This reduces communication in the online phase by a further 20%, and reduces round complexity, but adds some cost in an input-independent preprocessing phase.

LUT-256: S-box via a single table lookup. Our second class of protocols treats the S-box as a single lookup table of 256 elements. The main advantage of this approach is that it gets the best round complexity, since the S-box can be evaluated in a single round in the online phase (or 10 rounds for 1 AES block). Here, we present two different variants, based on either additive secret sharing ($\langle\langle\cdot\rangle\rangle$) or replicated secret sharing ($[\![\cdot]\!]$). Replicated secret sharing has the lowest online communication cost, but has very expensive preprocessing. With additive secret sharing, through a novel approach we are able to reduce the preprocessing cost by more than 10x, whilst only doubling the communication in the online phase. Based on our initial implementation of the semi-honest version of the replicated protocol, it seems that the LUT-256 protocols are best suited to a WAN setting, where round complexity is more critical, especially since the local computation cost of the table lookups is larger.

Achieving malicious security. One of the main challenges in our protocols is to achieve malicious security with a low overhead. One reason this is difficult is that our protocols rely on additive secret at various points, instead of purely a robust scheme like replicated or Shamir secret sharing. This makes it hard to apply standard verification techniques, such as the batch multiplication procedure of Boneh et al. [11], which is often used for distributed zero-knowledge proofs and MPC protocols [15, 32]. To overcome this, we carefully design our protocols such that the necessary replicated shares can be extracted from our additively shared table lookup protocols. This allows the result of a table lookup on additively shared inputs to be cheaply verified by checking the *previous* multiplication gate. Additionally, for our additively shared LUT-256 protocol, we rely on the algebraic structure of the S-box

Table 1: Performance comparison of our multi-party AES protocols and other approaches (communication measured in bits). For malicious security, our protocols incur an additional $O(\log N)$ rounds in both the preprocessing and online phases, where N is the total number of multiplications verified

† **the protocol communicates $O(\kappa)$ during the input phase but nothing is sent during the computation phase.**

	Preprocessing Phase		Online Phase			
Protocol	Comm.	Rounds	Comm.	Rounds	Total Comm.	Malicious
Obliv. select [43]	–	–	286720	30	39800	✗
$GF(2)$-circuit [6, 43]	–	–	5120	60	5120	✗
$GF(2)$-circuit (mal.) [5]	–	–	35840	60	35840	✓
$GF(2^8)$-circuit [21]	–	–	5120	30	5120	✗
$GF(2^8)$-circuit (mal.) [21, 31]	≈ 15000	$O(1)$	6144	30	≈ 21144	✓
Garbled circuit [44, 49]	614400	1	$0^{\dagger}$	1	≈ 614400	✓
$GF(2^4)$-circuit	–	–	3200	30	3200	✓
$GF(2^4)$-LUT-16	1760	2	2560	20	4320	✓
$\langle\langle\cdot\rangle\rangle$-LUT-256	3520	2	2560	10	6080	✓
$[\![\cdot]\!]$-LUT-256	39520	6	1280	10	39800	✓

to reduce the cost of the correctness check of an S-box computation, after a potentially faulty table lookup, inspired by recent work in zero-knowledge proofs [8]. This allows all of our maliciously secure protocols to have the same amortized communication cost as their semi-honest counterparts.

As a stepping stone in one of our protocols, we also obtain a general-purpose, malicious protocol for table lookups with $O(\sqrt{N})$ communication complexity for a table of size N. To the best of our knowledge, all prior practical approaches with malicious security require a cost of $\Omega(N)$. Our protocol relies on the observation that a table lookup can be verified using a single inner product check. By applying the generalised version of the batch multiplication verification from [11], which allows for checking inner product relations, we show how to verify a $O(\sqrt{N})$ complexity lookup table with almost no communication overhead.

Interestingly, for our additively shared AES protocol based on a specialization of this technique, we observe that it's not sufficient to directly use the verification protocol from [11, 15, 32], which inherently leak any errors in multiplication (or inner product) triples being verified to the adversary. While this leakage would typically be harmless, since the errors are already known to the adversary, this turns out not to be the case for us (for further discussion, see Section 3.5.2). We therefore show how to modify the verification procedure to remove any leakage.

Overall, as can be seen in Table 1, our maliciously secure protocols lead to a large reduction in communication costs compared with prior approaches. Our implementation results show that this approach comes with a slight increase in computational costs, but is still highly practical.

1.3 Related Work

1.3.1 Background. Known oblivious AES protocols are classified into two primary categories: those utilizing garbled circuits and those employing secret sharing schemes. Garbled circuit-based approaches [35, 37, 40, 49, 53] have the advantage of fewer communication rounds, but have high bandwidth costs due to the extensive size of garbled circuits. On the other hand, secret sharing schemes requires more communication rounds, but less communication. In recent years, efficient methods for performing secure Boolean operations have been proposed [6, 21] in the honest-majority setting, improving the performance of AES in MPC. The main challenge in constructing oblivious AES protocols lies in computing S-boxes, which requires non-linear operations [6] and thus communication. Previous research addressing this challenge can be broadly categorized into two types: methods using secure table lookup protocols [42, 43], and secure computation of algorithms that focus on the specific structure of S-boxes [23, 43]. Our proposed protocol integrates good aspects from both of these methods.

1.3.2 Oblivious AES using Lookup Tables. Oblivious AES computation using secure table lookup was proposed by Launchbury *et al.* [42] and Laur *et al.* [43], by converting the secret lookup index x into a one-hot vector encoding, with a one in position x and zeroes elsewhere. This approach performs the encoding entirely in the online phase, and requires 304 secure multiplications and 3 rounds to process one S-box as a size-256 table. Later works have used this technique in both the dishonest majority and honest majority settings [7, 18, 25, 40, 47], with the main improvement being to offload the computation of the one-hot vector to a preprocessing phase, leading to a very lightweight online phase. For example, the protocols from [18, 40] require $2^k - k - 1$ secure AND gates to preprocess a table of size $N = 2^k$. Our $[\![\cdot]\!]$-LUT-256 protocol is based on these ideas applied to the setting of replicated secret sharing. Other approaches using distributed point functions can reduce the bandwidth cost to $O(k\lambda)$, for security parameter λ, but this comes with computational security and an expensive setup phase [13, 14].

1.3.3 Structure of the S-box. Protocols exploiting the structure of the AES S-box have proposed various ways to improve efficiency. Laur *et al.* [43] securely computed S-boxes with the optimized Boolean circuit of Boyar *et al.* [12], obtaining a protocol in 6 rounds and with 32 AND gates. Subsequent works [5, 6] employ the same technique in the replicated secret-sharing setting for semi-honest and malicious adversaries. By focusing on the algebraic structure of the S-box, methods like the one in [21, 23, 24, 40] securely compute the S-box as a multiplicative inverse $x^{-1} = x^{254}$ in $GF(2^8)$. The previous most efficient AES protocols compute the inversion as a circuit in $GF(2^8)$ [21, 40]. In Chida *et al.*'s protocol [21], this inversion can be performed with only four secure multiplications in three rounds. While it is also possible to interpolate the AES S-box as a sparse polynomial in $GF(2^8)$ [45], as explored in [24], the cost of 18 multiplications in 12 rounds is prohibitively high compared to other techniques.

1.3.4 Technique for Multiplicative Inverse. The technique we use in this paper to calculate the multiplicative inverse using an extension of $GF(2^4)$ has mainly been used in hardware implementations of AES [51, 52]. Garbled circuit-based methods for oblivious AES have used optimized circuits based on this approach [37].

1.3.5 Maliciously Secure Protocols. There are also oblivious AES protocols that are secure against malicious adversaries [5, 24, 26, 27]. Our maliciously secure protocols target the three-party honest-majority setting. In the same setting, [5], improving over [31], to compute Boolean circuits use bucket cut-and-choose techniques with a total communication cost of 7 bits per AND gate. The three-party garbling framework by Mohassel et al. [44] lifts any semi-honest two-party garbling scheme into a malicious three-party protocol in the honest-majority setting. Instantiated with the Three-Halves scheme [49], the communication cost per AND gate is $1.5\kappa \approx 120$ bits for 80-bit security.

In other settings, maliciously secure two-party protocols [26] and multi-party protocols [23, 24, 27] for dishonest majority implement the AES function. These protocols require more than five times the communication cost compared to semi-honest secure protocols. Due to the new multiplication verification check, our maliciously secure protocols only have a logarithmic communication overhead over the semi-honest variants.

2 PRELIMINARIES

We begin by outlining some notation. We write $\vec{x}$ to denote vectors and index them as $\vec{x}_i$. We $\vec{x} \cdot \vec{y}$ to describe the inner product. We use $\vec{z} = \vec{x} \| \vec{y}$ to denote concatenation, i.e., $\vec{z}$ first contains elements of $\vec{x}$, then $\vec{y}$. One-hot vectors are written as $\boldsymbol{e}^{(r)}$ where r is the index of the single one in the vector, i.e., $\boldsymbol{e}_r^{(r)} = 1$ and $\boldsymbol{e}_i^{(r)} = 0$ for $i \neq r$. Public truth table vectors are denoted with T, omitting the $\vec{\cdot}$.

2.1 Finite Fields and Field Inversion

Let $\mathbb{F}_2$ be the field of order 2. We define the finite fields $GF(2^8)$ and $GF(2^4)$ as follows:

$$GF(2^8) := \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)\,,$$
$$GF(2^4) := \mathbb{F}_2[X]/(X^4 + X + 1)\,.$$

Note that the irreducible polynomial used in the definition of $GF(2^8)$ comes from the AES specification [28]. We consider elements of $GF(2^8)$ as bit-strings of length 8 corresponding to the coefficients of a degree-7 polynomial over $\mathbb{F}_2$. In particular, we write $\sum_{i=0}^{7} a_i x^i \in GF(2^8)$ as the string $\{a_7 a_6 \ldots a_1 a_0\}_2$ or the corresponding 2-digit hexadecimal notation. For example, $\{6E\}_{16}$ represents the equivalence class of $X^6 + X^5 + X^3 + X^2 + X$. Similarly, elements of $GF(2^4)$ are represented as 4-bit sequences or 1-digit hexadecimal notation. For example, $\{E\}_{16} \in GF(2^4)$ represents the equivalence class of $X^3 + X^2 + X$.

We define the finite field (extension) $GF((2^4)^2)$ over $GF(2^4)$ as

$$GF((2^4)^2) := GF(2^4)[X]/(X^2 + X + \{E\}_{16})\,.$$

Elements of $GF((2^4)^2)$ are represented as a degree-1 polynomial $a_h X + a_\ell$ over $GF(2^4)$. Each coefficient's binary representation is $\{a_{h3} a_{h2} a_{h1} a_{h0}\}_2$ and $\{a_{\ell 3} a_{\ell 2} a_{\ell 1} a_{\ell 0}\}_2$.

2.1.1 Multiplicative Inversion in $GF((2^4)^2)$. A formula for the inverse of $a = a_h X + a_\ell \in GF((2^4)^2)$ can be calculated by solving a system of equations derived from $aa^{-1} = 1$ (see, for instance, [20, 29]), giving

$$(a_h X + a_\ell)^{-1} = (a_h \otimes v^{-1})X + (a_h \oplus a_\ell) \otimes v^{-1}. \tag{1}$$

Here, $\oplus, \otimes$ represent addition and multiplication in $GF(2^4)$ respectively, and $v \in GF(2^4)$ is defined as follows

$$v := (a_h^2 \otimes \{E\}_{16}) \oplus (a_h \otimes a_\ell) \oplus a_\ell^2\ . \tag{2}$$

This shows that the inverse in $GF((2^4)^2)$ can be obtained through the calculation of one inverse v^{-1} in $GF(2^4)$, plus three multiplications and two squarings in $GF(2^4)$.

2.1.2 Isomorphism Between $GF(2^8)$ and $GF((2^4)^2)$. The finite fields $GF(2^8)$ and $GF((2^4)^2)$ are isomorphic. We use the explicit isomorphism and its inverse, described by Wolkerstorfer, Oswald and Lamberger [52], given by the following maps (specified in Appendix A):

$$\Phi\colon GF(2^8) \xrightarrow{\sim} GF((2^4)^2) : \{a_7 a_6 \ldots a_0\}_2 \longmapsto (a_h, a_\ell)\,,$$
$$\Phi^{-1}\colon GF((2^4)^2) \xrightarrow{\sim} GF(2^8) : (a_h, a_\ell) \longmapsto \{a_7 a_6 \ldots a_0\}_2\,.$$

2.2 Advanced Encryption Standard (AES)

AES is a block cipher standardized by NIST [28], with a 128-bit block size. We focus on AES-128, with a key length of 128 bits. For a detailed overview of the algorithm, we refer to Appendix A.

2.2.1 High-Level Structure and S-box. AES follows a substitution-permutation network design, with alternating layers of non-linear S-boxes and linear permutations. In addition, there is a key schedule that expands the 128-bit key into a set of round keys to be used in each round. The linear components of a round are ShiftRows, MixColumns and AddRoundKey, which XORs the state with the round key. These can be expressed as linear operations in $GF(2^8)$.

The S-box of AES — also called SubBytes — operates on one byte of the state at a time, and is the only non-linear part of AES. It can be expressed as the mapping over the finite field $GF(2^8)$ that sends $x \mapsto \mathsf{Affine}(x^{254})$, where Affine as an invertible affine transformation. Since the multiplicative group of $GF(2^8)$ has order 255, the computation of x^{254} maps every non-zero x to x^{-1} and 0 to 0. We will often abuse terminology slightly and refer to this as an inversion in $GF(2^8)$.

2.3 Security Model

We consider protocols secure against up to $t-1$ out of n corrupted parties, where $t - 1 < n/2$. We will often focus on the case of 1-out-of-3 corruptions, where $n = 3, t = 2$. All of our protocols are presented and analyzed in the malicious model with abort, where a corrupt party may deviate from the protocol specification and honest parties are not guaranteed to receive output. We also consider relaxations in the semi-honest model, where each party is assumed to follow the protocol, which are obtained by omitting any verification steps in the protocol.

We model ideal functionalities and give security proofs in the Universal Composability (UC) framework [19], which gives strong composition guarantees.

2.4 Secure Multi-Party Computation

We build upon MPC protocols based on replicated secret sharing [38]. This approach is well-suited for secure computations involving $GF(2^k)$ values such as those occurring in oblivious AES. Additionally, it offers the advantage of a lightweight protocol for multiplication of secret-shared values.

2.4.1 Replicated Secret Sharing and Additive Secret Sharing. We use both t-out-of-n replicated secret sharing ((t,n)-RSS) and n-out-of-n additive secret sharing ((n,n)-SS):

- $[\![a]\!]$: (t,n)-RSS. For a secret a in a finite field $\mathbb{F}$, a is split into $\binom{n}{t-1}$ random shares $a^{(T)} \in \mathbb{F}$, for each subset $T \subset [n]$ of size $t-1$, with the shares sampled so that $a = \sum_T a^{(T)}$. Party i gets the $\binom{n-1}{t-1}$ shares $[\![a]\!]_i = \{a^{(T)}\}_{T, i \notin T}$. The overall sharing of a is $[\![a]\!] = ([\![a]\!]_1, \ldots, [\![a]\!]_n)$. In case of three parties, each party holds exactly two of the three shares.
- $\langle\!\langle a \rangle\!\rangle$: (n,n)-SS. For a secret $a \in \mathbb{F}$, each party i holds $\langle\!\langle a \rangle\!\rangle_i = a^{(i)} \in \mathbb{F}$ s.t. $a = a^{(1)} + \cdots + a^{(n)}$. A share of a value a is $\langle\!\langle a \rangle\!\rangle = (\langle\!\langle a \rangle\!\rangle_1, \ldots, \langle\!\langle a \rangle\!\rangle_n)$.

In this paper, the field $\mathbb{F}$ is always characteristic 2, and typically either $GF(2)$, $GF(2^4)$, or $GF(2^8)$. With both replicated and additive sharing, a sharing in $GF(2^k)$ can be decomposed into an array of k shares over $GF(2)$ without communication by doing binary decomposition.

We denote the set of corrupted parties by C, and the set of honest parties by $\mathcal{H}$. For a sharing $[\![x]\!]$, we let $[\![x]\!]^C = \{[\![x]\!]_i\}_{i \in C}$ be the set of all corrupt parties' shares, and $[\![x]\!]^{\mathcal{H}}$ the set of all honest shares (and similarly define $\langle\!\langle \cdot \rangle\!\rangle^C$, $\langle\!\langle \cdot \rangle\!\rangle^{\mathcal{H}}$ for additive sharings).

Definition 2.1 (Consistent shares). Let $S \subset [n]$ and consider a set of replicated shares $\{[\![x]\!]_i\}_{i \in S}$, where $[\![x]\!]_i = \{x_i^{(T)}\}_{T \not\ni i}$. We say that the set of shares is *consistent* if $x_i^{(T)} = x_{i'}^{(T)}$ for every i, i' and T where $i, i' \notin T$.

When modelling security, we often define ideal functionalities where the adversary provides as input a set of shares $[\![x]\!]^C$. In this case, we implicitly require that the functionality only accepts a set of consistent shares. Our functionalities also often rely on the following straightforward fact.

PROPOSITION 2.2. *Given a secret x and a consistent set of corrupted parties' shares $[\![x]\!]^C$, a consistent set of honest shares $[\![x]\!]^{\mathcal{H}}$ can always be defined.*

Note that if exactly t parties are corrupted, then the honest parties' shares are defined uniquely; otherwise, they can be sampled at random.

2.4.2 Reconstruction. We consider reconstruction as an interactive protocol, where parties exchange shares and reconstruct a secret. We have the following two protocols.

Opening additive shares: $x = \mathsf{Reconst}(\langle\!\langle x \rangle\!\rangle)$. Each party sends its share $x^{(i)}$ to all other parties, and reconstructs $x = \sum x^{(i)}$. This requires $n-1$ field elements of communication per party in one round. Alternatively, one can use the "king" approach, where the parties send their shares to a designated party, who reconstructs and sends back x. This takes on average only $2(n-1)/n$ field elements per party but two rounds of interaction. In our implementation, since we focus on the 3-party setting, we chose to take the one-round approach with two elements of communication.

Note that in the malicious setting, a corrupted party can easily change the result of reconstruction by lying about their share.

Opening replicated shares: $x = \mathsf{Reconst}([\![x]\!])$. With replicated secret sharing, the parties can *robustly* open a secret, guaranteeing that each party either outputs the correct value, or aborts. The simplest protocol is for the parties to exchange all of their shares, and check whether the resulting sharing is consistent before reconstructing x. When reconstructing many values, this protocol can be optimized by optimistically sending the minimal number of shares needed to reconstruct x, and later verifying all openings in a batch by exchanging and comparing hashes of the remaining shares. This was demonstrated for the 3-party setting in [31] and later extended to the multi-party setting [41].

2.4.3 Correlated Randomness and Coin Tossing. We require the parties to have access to different forms of correlated randomness, for obtaining replicated sharings of random values, and additive sharings of zero (see Fig. 1). The $\mathcal{F}_{\mathsf{rand}}$ functionality can be implemented using pseudorandom secret sharing [22], where after a one-time setup to distribute pseudorandom function keys among the parties, the correlated randomness can be generated non-interactively. Obtaining random additive sharings of zero, modelled in $\mathcal{F}_{\mathsf{zero}}$, can similarly be done using pseudorandom secret sharing. One simple approach is to use $\mathcal{F}_{\mathsf{rand}}$ to obtain a random $[\![r]\!]$, and then each party XORs together $n-1$ of its share elements, appropriately selected such that all shares cancel out and sum to zero.

Finally, we also rely on the coin-tossing functionality, $\mathcal{F}_{\mathsf{coin}}$, which can be realized by running Reconst on a random sharing from $\mathcal{F}_{\mathsf{rand}}$.

2.4.4 Computations on Shares.

Linear Operations. Any $GF(2)$-linear operation can be performed locally on replicated or additively shared values, by simply applying the operation to each element of each party's share. Similarly, addition by a constant can be performed by adding it to a fixed subset of the share elements. Given sharings $[\![x]\!]$, $[\![y]\!]$ and public values a, c, we denote these operations by $[\![ax + y + c]\!] := a[\![x]\!] + [\![y]\!] + c$ and similarly $\langle\!\langle ax + y + c \rangle\!\rangle := a\langle\!\langle x \rangle\!\rangle + \langle\!\langle y \rangle\!\rangle + c$.

Free Squaring. Since squaring in $GF(2^k)$ is linear over $GF(2)$, it can be performed locally in both (t, n)$-RSS$ and (n, n)$-SS$: each party simply squares their corresponding shares. We denote $[\![x^2]\!] := \mathsf{Square}([\![x]\!])$.

Functionality $\mathcal{F}_{\mathsf{rand}}(\mathbb{F})$

(1) $\mathcal{F}_{\mathsf{rand}}$ receives from the adversary the shares $[\![r]\!]^{C}$.
(2) $\mathcal{F}_{\mathsf{rand}}$ samples $r \leftarrow \mathbb{F}$, and uses $(r, [\![r]\!]^{C})$ to define $[\![r]\!]^{\mathcal{H}}$.
(3) $\mathcal{F}_{\mathsf{rand}}$ distributes the shares $[\![r]\!]^{\mathcal{H}}$ to the honest parties.

Functionality $\mathcal{F}_{\mathsf{zero}}(\mathbb{F})$

(1) $\mathcal{F}_{\mathsf{zero}}$ receives from the adversary the shares $\langle\!\langle z \rangle\!\rangle^{C}$, and samples random shares $\langle\!\langle z \rangle\!\rangle^{\mathcal{H}}$, such that $z = 0 \in \mathbb{F}$.
(2) $\mathcal{F}_{\mathsf{zero}}$ distributes the shares $\langle\!\langle z \rangle\!\rangle^{\mathcal{H}}$ to the honest parties.

Functionality $\mathcal{F}_{\mathsf{coin}}(k)$

(1) $\mathcal{F}_{\mathsf{coin}}$ samples $(b_1, \ldots, b_k) \leftarrow \{0,1\}^k$ and sends this to the adversary.
(2) On receiving *OK* from the adversary, it delivers $(b_1, \ldots, b_k)$ to the honest parties.

Figure 1: The functionalities $\mathcal{F}_{\mathsf{rand}}$, $\mathcal{F}_{\mathsf{zero}}$ and $\mathcal{F}_{\mathsf{coin}}$.

Local Multiplication. We rely on the multiplicative property of replicated secret sharing: given sharings $[\![x]\!]$ and $[\![y]\!]$, the parties can obtain an additive sharing $\langle\!\langle xy \rangle\!\rangle$ *without any interaction*. This holds because when $t-1 < n/2$, any pair of share elements $x^{(T)}, x^{(T')}$, for size-$t-1$ subsets T, T', is held by at least one party. For example, when $n = 3$, party i holds $(x^{(i)}, x^{(i+1)})$, $(y^{(i)}, y^{(i+1)})$ and can compute $\langle\!\langle xy \rangle\!\rangle_i := x^{(i)}y^{(i)} + (x^{(i)} + x^{(i+1)})(y^{(i)} + y^{(i+1)})$. We write this multiplication as $\langle\!\langle xy \rangle\!\rangle := \mathsf{MultLocal}([\![x]\!], [\![y]\!])$.

Share Conversion from $\langle\!\langle x \rangle\!\rangle$ to $[\![x]\!]$. This can be achieved with a single communication round. Essentially, each party creates a replicated secret sharing $[\![x^{(i)}]\!]$ of its additive share, and distributes the resulting shares to the remaining parties. With $n = 3$, this can be achieved by having the parties first re-randomize their shares by adding a share of zero from $\mathcal{F}_{\mathsf{zero}}$. Then, party i sends its share $x^{(i)}$ to party $i+1$. The cost is sending 1 field element per party in one round [6]. We write this as $[\![x]\!] = \mathsf{Reshare}(\langle\!\langle x \rangle\!\rangle)$.

Multiplication with Additive Errors. Combining the local multiplication and resharing protocols, above, we obtain a multiplication protocol on $[\![\cdot]\!]$-shared values. In the malicious setting, a corrupted party may cheat during the resharing step, introducing an error into the output. We model this in the $\mathcal{F}_{\mathsf{weakMult}}$ functionality (Fig. 2), adapted from [33], which allows the adversary to choose an error d that is added into the output shares. We additionally extend this to $\mathcal{F}_{\mathsf{weakDotProduct}}$, for computing an inner product, where the inner product is computed locally followed by a single resharing.

2.5 Verifying Multiplications and Dot Products with Malicious Security

To ensure correct multiplications with malicious security, we use a batch verification procedure. The idea is that during the main MPC execution, the parties can use $\mathcal{F}_{\mathsf{weakMult}}$, and later verify that all multiplications were correct with batch verification. This batch verification can be safely postponed until the end of the protocol, as long as any outputs of the computation are only revealed *after* the multiplications have been verified.

Functionality $\mathcal{F}_{\mathsf{weakMult}}$ / $\mathcal{F}_{\mathsf{weakDotProduct}}$

Input: $[\![\vec{x}]\!], [\![\vec{y}]\!]$: (t,n)-RSS share of $\vec{x}$ and $\vec{y}$.
Output: $[\![\vec{x} \cdot \vec{y}]\!]$: (t,n)-RSS share of the product $\vec{x} \cdot \vec{y}$.

(1) $\mathcal{F}_{\mathsf{weakMult}}$ receives the shares $[\![\vec{x}]\!]^{\mathcal{H}}, [\![\vec{y}]\!]^{\mathcal{H}}$ from honest parties and reconstructs the secrets $\vec{x}, \vec{y}$.
(2) $\mathcal{F}_{\mathsf{weakMult}}$ computes the corrupted parties' shares $[\![\vec{x}]\!]^{C}, [\![\vec{y}]\!]^{C}$ and sends these shares to the adversary.
(3) $\mathcal{F}_{\mathsf{weakMult}}$ receives an error d and a set of shares $[\![z]\!]^{C}$ from the adversary.
(4) $\mathcal{F}_{\mathsf{weakMult}}$ computes $z := \vec{x} \cdot \vec{y} + d$ and samples honest parties' shares $[\![z]\!]^{\mathcal{H}}$ by using d and $[\![z]\!]^{C}$.
(5) $\mathcal{F}_{\mathsf{weakMult}}$ distributes the shares of $[\![z]\!]^{C}$ to honest parties.

Figure 2: Functionality for multiplication with additive error. We refer to the functionality as $\mathcal{F}_{\mathsf{weakMult}}$ if $\vec{x}, \vec{y}$ have length-1, and $\mathcal{F}_{\mathsf{weakDotProduct}}$ otherwise.

Prior works implementing MPC for Boolean circuits, such as [5, 31], use cut-and-choose techniques to verify multiplications. These have a large communication overhead, and require running in very large batches to obtain reasonable parameters. Instead, we verify multiplication triples by adapting the protocol of [32, 33], originally presented for Shamir-based MPC, and based on similar ideas used previously for distributed zero-knowledge proofs [11] and MPC [15, 16, 46].

Adapting the protocol to the setting of replicated secret sharing, with security with abort, is quite straightforward. However, we make two key changes that are needed in some of our applications. Firstly, we extend the protocol to verifying a batch of inner product relations, rather than just multiplications. This allows for a maliciously secure inner product protocol with communication independent of the vector size, apart from the batch verification procedure which only scales logarithmically with the *total* length of all inner products. Secondly, prior works [16, 32] only realized a functionality that leaks the errors $z_i - x_i y_i$ in all multiplication triples to the adversary. Instead, we realize the functionality $\mathcal{F}_{\mathsf{verify}}$ (Fig. 3), which *only* leaks the result of the verification check, and no additional information. This turned out to be critical for proving security of our AES protocol in Section 3.5.3.

The protocol is shown in Protocol 2. To ensure correct triples with high probability, we need to work over an exponentially large finite field $\mathbb{F}$; we therefore use the protocol by first taking our Boolean (or small field) triples and lifting the shares into a large extension field. Then, the protocol begins by randomizing the batch of inner product triples, converting it into a single, large inner product of length N. To verify the inner product, the protocol proceeds in $\log N$ rounds, where in each round the dimension is halved, by first viewing the inner product as an inner product on length-$N/2$ vectors of suitably defined degree-1 polynomials, followed by evaluting the polynomials at a random challenge to compress this to an inner product of vectors of field elements. Eventually, it reaches a base case where $N = 1$, and performs a naive triple check (Protocol 1) that uses one extra, random multiplication and a random

Protocol 1 CheckTriple($[\![x]\!], [\![y]\!], [\![z]\!]$) $\rightarrow 0/1$

```
Input: $[\![x]\!], [\![y]\!], [\![z]\!]$.
Output: Verify that $xy = z$.
1: $[\![x']\!], [\![r]\!] \leftarrow \mathcal{F}_{\mathsf{rand}}(\mathbb{F})$
2: $[\![z']\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![x']\!], [\![y]\!])$
3: $t \leftarrow \mathcal{F}_{\mathsf{coin}}(\mathbb{F})$
4: $\rho := \mathsf{Reconst}([\![x]\!] + t[\![x']\!])$
5: $[\![\sigma]\!] := \mathcal{F}_{\mathsf{weakMult}}([\![z]\!] + t[\![z']\!] - \rho[\![y]\!], [\![r]\!])$
6: if Reconst($[\![\sigma]\!]$) = 0 then
7:     return 1
8: else
9:     return 0
10: end if
```

Functionality $\mathcal{F}_{\mathsf{verify}}$

Input: $([\![\vec{x}^{(1)}]\!], [\![\vec{y}^{(1)}]\!], [\![z^{(1)}]\!]), \ldots, ([\![\vec{x}^{(m)}]\!], [\![\vec{y}^{(m)}]\!], [\![z^{(m)}]\!])$, where m is the number of inner product triples to be verified.

Output: $b \in \{\mathsf{accept}(1), \mathsf{abort}(0)\}$ to honest parties.

(1) $\mathcal{F}_{\mathsf{verify}}$ receives from honest parties their shares of $([\![\vec{x}^{(i)}]\!], [\![\vec{y}^{(i)}]\!], [\![z^{(i)}]\!])$ to reconstruct $(\vec{x}^{(i)}, \vec{y}^{(i)}, z^{(i)})$ for all $i \in [m]$.

(2) $\mathcal{F}_{\mathsf{verify}}$ computes the rest of the corrupted parties' shares of $([\![\vec{x}^{(i)}]\!], [\![\vec{y}^{(i)}]\!], [\![z^{(i)}]\!])$ for all $i \in [m]$ and sends these shares to the adversary.

(3) $\mathcal{F}_{\mathsf{verify}}$ computes $d^{(i)} = z^{(i)} - \vec{x}^{(i)} \cdot \vec{y}^{(i)}$.

(4) $\mathcal{F}_{\mathsf{verify}}$ sets $b := \mathsf{abort}$ if there exists $i \in [m]$ such that $d^{(i)} \neq 0$ and sets $b := \mathsf{accept}$ otherwise.

(5) $\mathcal{F}_{\mathsf{verify}}$ sends b to the adversary and proceeds as follows:
- If the adversary replies continue, send b to honest parties.
- If the adversary replies abort, send abort to honest parties.

Figure 3: The functionality $\mathcal{F}_{\mathsf{verify}}$.

challenge to check the remaining one. Importantly, step 5 of the protocol computes the value $z + tz' - \rho y = (z - xy) + t(z' - x'y)$, which should equal 0 if the triple is correct. However, this cannot be revealed directly, as it would leak information on $z - xy$ to the adversary. To prevent this, we rerandomize it via the additional multiplication with $[\![r]\!]$; this change allows us to realize the stronger $\mathcal{F}_{\mathsf{verify}}$ functionality.

We prove the following in Appendix B.

THEOREM 2.3. *Protocol 2 (*Verify*) securely realizes the functionality $\mathcal{F}_{\mathsf{verify}}$ (see Fig. 3), in the $(\mathcal{F}_{\mathsf{weakMult}}, \mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{rand}})$-hybrid model. The failure probability in the simulation is at most $(m+2\log N)/|\mathbb{F}|$.*

2.6 Functionality for Table Lookup

In Fig. 4, we present a functionality for performing a secret-shared table lookup. The functionality is parameterized by the secret-sharing schemes used for the inputs and outputs, denoted ss_1 and ss_2 respectively, which can be any combination of replicated shares ($[\![\cdot]\!]$)

Protocol 2 Verifying a batch of inner products

```
Functionality: $0/1 \leftarrow \mathcal{F}_{\mathsf{verify}}([\![\cdots]\!])$.
Input: Shared triples $\{[\![\vec{x}_i]\!], [\![\vec{y}_i]\!], [\![z_i]\!]\}_{i=0}^{m-1}$, where $\vec{x}_i, \vec{y}_i \in \mathbb{F}^{n_i}$ and $z_i \in \mathbb{F}$ for each $i$.
Output: Verify that $\vec{x}_i \cdot \vec{y}_i = z_i$, for all $i$.
1: $r \leftarrow \mathcal{F}_{\mathsf{coin}}(\mathbb{F})$
2: return VerifyDotProduct($[\![\vec{x}_0]\!], r[\![\vec{x}_1]\!], \cdots, r^{m-1}[\![\vec{x}_{m-1}]\!], [\![\vec{y}_0]\!], \cdots, [\![\vec{y}_{m-1}]\!], \sum_{i=0}^{m-1} r^i [\![z_i]\!]$)
3: return VerifyDotProduct($[\![\vec{x}_0 \| r\vec{x}_1 \| \cdots \| r^{m-1}\vec{x}_{m-1}]\!], [\![\vec{y}_0 \| \cdots \| \vec{y}_{m-1}]\!], \sum_{i=0}^{m-1} r^i [\![z_i]\!]$)

4: procedure VerifyDotProduct(($[\![x_0]\!], \ldots, [\![x_{N-1}]\!]$), ($[\![y_0]\!], \ldots, [\![y_{N-1}]\!]$), $[\![z]\!]$)
5:     if $N = 1$ then
6:         return CheckTriple($[\![x_0]\!], [\![y_0]\!], [\![z_0]\!]$)
7:     end if

8:     for $i = 0$ to $N/2 - 1$ do
9:         $[\![f_i(X)]\!] := [\![x_{2i}]\!] + ([\![x_{2i}]\!] + [\![x_{2i+1}]\!])X$
10:        $[\![g_i(X)]\!] := [\![y_{2i}]\!] + ([\![y_{2i}]\!] + [\![y_{2i+1}]\!])X$
11:    end for        ▷polynomials in $X$ such that $f_i(b) = x_{2i+b}$, $g_i(b) = y_{2i+b}$, for $b \in \{0, 1\}$
12:    $[\![h(1)]\!] \leftarrow \mathcal{F}_{\mathsf{weakDotProduct}}([\![\vec{f}(1)]\!], [\![\vec{g}(1)]\!])$
13:    $[\![h(2)]\!] \leftarrow \mathcal{F}_{\mathsf{weakDotProduct}}([\![\vec{f}(2)]\!], [\![\vec{g}(2)]\!])$
14:    $[\![h(0)]\!] := [\![z]\!] - [\![h(1)]\!]$        ▷defines degree-2 $h(X)$
15:    $r \leftarrow \mathcal{F}_{\mathsf{coin}}(\mathbb{F})$
16:    Locally compute $[\![h(r)]\!]$ via Lagrange interpolation
17:    return VerifyDotProduct($[\![\vec{f}(r)]\!], [\![\vec{g}(r)]\!], [\![h(r)]\!]$)
18: end procedure
```

and additive shares ($\langle\!\langle\cdot\rangle\!\rangle$). We shorten the description to $\mathcal{F}_{\mathsf{LUT}}^{ss}$ if $ss_1 = ss_2$. Note that the variant $[\![\cdot]\!] \mapsto [\![\cdot]\!]$ is fully maliciously secure, while variants where the input or output is $\langle\!\langle\cdot\rangle\!\rangle$-shared inherently allow a corrupt party to change the input or output. Importantly, if the input is $\langle\!\langle\cdot\rangle\!\rangle$-shared then the functionality additionally outputs a $[\![\cdot]\!]$ sharing of the input that was used. We use this in our maliciously secure protocols for verifying the correct inputs were used after the protocol execution.

In Section 3.3, we will describe a protocol for realizing $\mathcal{F}_{\mathsf{LUT}}^{\langle\!\langle\cdot\rangle\!\rangle \rightarrow [\![\cdot]\!]}$. Later, in Section 3.5.3, we give protocols for the other variants.

3 PROTOCOLS FOR MULTI-PARTY AES

In this section, we construct the MPC protocols that compute AES. We define the ideal functionality $\mathcal{F}_{\mathsf{AES}}$ as the functionality that computes the AES block encryption taking secret-shared inputs $\{[\![x_i]\!]\}_{i=0,\ldots,127}$ and a secret-shared encryption key $\{[\![k_i]\!]\}_{i=0,\ldots,127}$, and returns shared outputs of $\mathsf{Enc}(k, x)$, $\{[\![z_i]\!]\}_{i=0,\ldots,127}$. All proposed protocols securely compute the ideal functionality $\mathcal{F}_{\mathsf{AES}}$.

3.1 Overview of the Proposed Protocols

To compute the AES algorithm in MPC, all steps need to be computed on secret-shared data. However, the linear layers of AES (ShiftRows, MixColumns, AddRoundKey) can be computed locally

Functionality $\mathcal{F}_{\mathsf{LUT}}^{ss_1 \mapsto ss_2} / \mathcal{F}_{\mathsf{LUT}}^{ss}$

Input: v shared under scheme $ss_1 \in \{\langle\langle\cdot\rangle\rangle, [\![\cdot]\!]\}$, and public vector T.

Output: T_v shared under scheme $ss_2 \in \{\langle\langle\cdot\rangle\rangle, [\![\cdot]\!]\}$, where T_v is the v-th element of T. If $ss_1 = \langle\langle\cdot\rangle\rangle$, also output $[\![v]\!]$.

(1) To define the input sharings, if $ss_1 = \langle\langle\cdot\rangle\rangle$:
- $\mathcal{F}_{\mathsf{LUT}}$ receives from each party a share $\langle\langle v\rangle\rangle^i$, and reconstructs $v = \sum_i \langle\langle v\rangle\rangle^i$.

(2) Otherwise, if $ss_1 = [\![\cdot]\!]$:
- $\mathcal{F}_{\mathsf{LUT}}$ receives from honest parties their shares $[\![v]\!]^{\mathcal{H}}$, and reconstructs v.
- $\mathcal{F}_{\mathsf{LUT}}$ computes the corrupted parties' shares $[\![v]\!]^{C}$ and sends these to the adversary.

(3) $\mathcal{F}_{\mathsf{LUT}}$ looks up T_v.

(4) $\mathcal{F}_{\mathsf{LUT}}$ receives from the adversary a set of corrupted shares $\mathsf{sh}(T_v)^C$, under scheme ss_2.

(5) $\mathcal{F}_{\mathsf{LUT}}$ samples consistent honest shares $\mathsf{sh}(T_v)^{\mathcal{H}}$ using $(T_v, \mathsf{sh}(T_v)^C)$, under scheme ss_2.

(6) If $ss_1 = \langle\langle\cdot\rangle\rangle$: $\mathcal{F}_{\mathsf{LUT}}$ additionally receives consistent shares $[\![v]\!]^C$ from the adversary, and defines the honest shares $[\![v]\!]^{\mathcal{H}}$ using $(v, [\![v]\!]^C)$.

(7) $\mathcal{F}_{\mathsf{LUT}}$ outputs the shares $\mathsf{sh}(T_v)^{\mathcal{H}}$, and optionally $[\![v]\!]^{\mathcal{H}}$, to the honest parties.

Figure 4: Ideal functionality for secret-shared table lookup.

on the shares, as detailed in Sect. 2.4.4. Thus, we focus on constructing the non-linear operations within the KeyExpansion and SubBytes steps, that is, multiplicative inversions in $GF(2^8)$ within the S-box. This is (essentially) the only place where our variants differ. For completeness, we give the full oblivious AES algorithm in Protocol 8 in Appendix A. We briefly summarize the following approaches to compute multiplicative inversions in $GF(2^8)$; their costs are also summarized in Table 2.

The straightforward approach is to compute the inverse directly using a 256-elements lookup table. We call this protocol the LUT-256 (see Sect. 3.5). Here, parties prepare a public 256-elements lookup table for inversion and convert an input share $[\![x]\!]$ for $x \in GF(2^8)$ into a one-hot vector $\boldsymbol{e}^{(x)} \in \{0,1\}^{256}$ that has 1 at the position x and 0 otherwise. Then, they compute an inner product of the lookup table and the one-hot vector to obtain $[\![x^{-1}]\!]$. While this approach is optimal in round complexity, it requires heavy offline communication to generate random one-hot vectors.

The main protocol (Protocol 3 in Sect. 3.2), LUT-16, reduces such offline cost by using a smaller lookup table. The idea is to reduce the computation of multiplicative inversion over $GF(2^8)$ into the one over $GF(2^4)$ as shown in Eq. (1), (2) by using the isomorphism between $GF(2^8)$ and $GF((2^4)^2)$ from Sect. 2.1.2. Computing the inverse in $GF(2^4)$ only requires a look-up table of size 16.

Our lookup table protocol (Protocol 4) in Sect. 3.3 takes $\langle\langle v\rangle\rangle$, $v \in GF(2^4)$ as well as a public table T and outputs the corresponding shared value $[\![T_v]\!]$ using the table. When the table is the inversion table, the obtained output $T_x = v^{-1}$. The key subfunctionality is $\mathcal{F}_{\mathsf{RandOHV}}$ by which we can obtain a shared randomness $[\![r]\!]$ and the corresponding shared one-hot vector $[\![\boldsymbol{e}^{(r)}]\!]$. This allows the table lookup to be performed as a linear function on $\boldsymbol{e}^{(r)}$, after reconstructing a masked value $c = v \oplus r$ in the online phase, for a lookup table input v.

Table 2: Comparison of maliciously secure S-Box evaluation methods. # Mult is the number of multiplication triples checked by $\mathcal{F}_{\mathsf{verify}}$.

	Offline Comm.	Online Comm.	Rounds	# Mult
Boolean circuit	–	32	6	32
$GF(2^8)$ circuit	–	32	4	4
$GF(2^4)$ circuit	–	20	4	5
$GF(2^4)$/LUT-16	11	16	2	3
$(3,3)$ LUT-256	22	16	1	7
$(2,3)$ LUT-256	247	8	1	7

Functionality $\mathcal{F}_{\mathsf{Inv}}$

Input: $[\![x]\!]$

Output: $[\![x^{-1}]\!]$

(1) $\mathcal{F}_{\mathsf{Inv}}$ receives from honest parties their shares of $[\![x]\!]^{\mathcal{H}}$ to reconstruct x.

(2) $\mathcal{F}_{\mathsf{Inv}}$ computes the corrupted parties' shares $[\![x]\!]^{C}$ and sends these shares to the adversary.

(3) $\mathcal{F}_{\mathsf{Inv}}$ obtains x^{-1} from x.

(4) $\mathcal{F}_{\mathsf{Inv}}$ receives a set of shares $[\![x^{-1}]\!]^C$ from the adversary.

(5) $\mathcal{F}_{\mathsf{Inv}}$ computes the honest shares $[\![x^{-1}]\!]^{\mathcal{H}}$ by using the set of shares $[\![x^{-1}]\!]^C$ and x^{-1}.

(6) $\mathcal{F}_{\mathsf{Inv}}$ outputs the shares $[\![x^{-1}]\!]^{\mathcal{H}}$ to the honest parties.

Figure 5: The functionality $\mathcal{F}_{\mathsf{Inv}}$ for $GF(2^8)$.

3.2 Secure Protocol for Multiplicative Inverse

We propose a protocol for securely computing the multiplicative inverse in $GF(2^8)$ in Protocol 3. The ideal functionality $\mathcal{F}_{\mathsf{Inv}}$ is described in Fig. 5. By applying Φ from Sect. 2.1.1, the secure computation of a multiplicative inverse in $GF(2^8)$ can be reduced to three secure multiplications in $GF(2^4)$ and one secure multiplicative inverse in $GF(2^4)$.

To achieve malicious security, we need to ensure that corrupt parties use the correct additively shared input $\langle\langle v\rangle\rangle$ to $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle \to [\![\cdot]\!]}$. To do this, we rely on the fact that $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle \to [\![\cdot]\!]}$ also outputs the replicated sharing $[\![v]\!]$, giving a commitment to what value was actually used as input. We then check that v was correct by working backwards until the previous multiplication (step 6), computing replicated shares of the multiplication input. Then, if we verify this multiplication using $\mathcal{F}_{\mathsf{verify}}$, this guarantees that the correct value was input to $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle \to [\![\cdot]\!]}$.

We prove the following in Appendix D.1.

Lemma 3.1. *The protocol* Inv *in Protocol 3 securely computes* $\mathcal{F}_{\mathsf{Inv}}$ *with abort in the* $\left\{\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle \to [\![\cdot]\!]}, \mathcal{F}_{\mathsf{weakMult}}, \mathcal{F}_{\mathsf{verify}}\right\}$*-hybrid model in*

Protocol 3 Multiplicative Inversion over $GF(2^8)$ against Malicious Adversary

Functionality: $[\![x^{-1}]\!] \leftarrow \mathcal{F}_{\mathsf{Inv}}([\![x]\!])$
Input: Share $[\![x]\!]$ of $x \in GF(2^8)$
Output: Share $[\![x^{-1}]\!]$ of the inverse of $x \in GF(2^8)$
Subfunctionality: LUT

```
1: $([\![a_h]\!], [\![a_\ell]\!]) \leftarrow \Phi([\![x]\!]) \in GF((2^4)^2)$    ▷Φ in Sec. 2.1.2
2: $[\![a_h^2]\!] := \mathsf{Square}([\![a_h]\!])$
3: $\langle\!\langle a_h^2 \rangle\!\rangle \leftarrow \mathsf{ToAdditive}([\![a_h^2]\!])$
4: $[\![a_\ell^2]\!] := \mathsf{Square}([\![a_\ell]\!])$
5: $\langle\!\langle a_\ell^2 \rangle\!\rangle := \mathsf{ToAdditive}([\![a_\ell^2]\!])$
6: $\langle\!\langle a_h \times a_\ell \rangle\!\rangle := \mathsf{MultLocal}([\![a_h]\!], [\![a_\ell]\!])$
7: $\langle\!\langle v \rangle\!\rangle := (\{E\}_{16} \times \langle\!\langle a_h^2 \rangle\!\rangle) \oplus \langle\!\langle a_h \times a_\ell \rangle\!\rangle \oplus \langle\!\langle a_\ell^2 \rangle\!\rangle$
8: $([\![v^{-1}]\!], [\![v]\!]) \leftarrow \mathcal{F}_{\mathsf{LUT}}^{\langle\!\langle\cdot\rangle\!\rangle \to [\![\cdot]\!]}(\langle\!\langle v \rangle\!\rangle, T^{inv})$    ▷1 round, 8 bits
9: $[\![a_h \times a_\ell]\!] := [\![v]\!] \oplus (\{E\}_{16} \times [\![a_h^2]\!]) \oplus [\![a_\ell^2]\!]$ ▷Necessary for $\mathcal{F}_{\mathsf{verify}}$
10: $[\![a'_h]\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![a_h]\!], [\![v^{-1}]\!])$
11: $[\![a'_\ell]\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![a_h]\!] \oplus [\![a_\ell]\!], [\![v^{-1}]\!])$    ▷1 round, 8 bits
12: $[\![y]\!] \leftarrow \Phi^{-1}([\![a'_h]\!], [\![a'_\ell]\!]) \in GF(2^8)$
13: Execute $\mathcal{F}_{\mathsf{verify}}$ for the following multiplication triplets:
    (1) $([\![a_h]\!], [\![a_\ell]\!], [\![a_h \times a_\ell]\!])$
    (2) $([\![a_h]\!], [\![v^{-1}]\!], [\![a'_h]\!])$
    (3) $([\![a_h \oplus a_\ell]\!], [\![v^{-1}]\!], [\![a'_\ell]\!])$
14: return $[\![y]\!]$
```

the presence of a malicious adversary under the honest majority setting.

Reducing the number of multiplication checks. Instead of checking 3 multiplications, we observe that the protocol can be optimized by embedding all checks into one multiplication. For some $k \geq 3$ and degree-k, irreducible polynomial $f(\alpha)$ over $GF(2^4)$, define the extension field $K = GF((2^4)^k) = GF(2^4)[\alpha]/f(\alpha)$. An element of K can be expressed as a polynomial $a_{k-1}\alpha^{k-1} + \cdots + a_1\alpha + a_0$, for $a_i \in GF(2^4)$. Then, we can check the following equation over K:

$$([\![a_h]\!]+\alpha[\![a_\ell]\!])\cdot([\![a_\ell]\!]+\alpha[\![v^{-1}]\!]) = [\![a_h \times a_\ell]\!]+\alpha[\![a'_h + a_\ell^2]\!]+\alpha^2[\![a'_\ell \oplus a'_h]\!] .$$

Note that shares of a_ℓ^2 can be computed locally. Furthermore, notice that if the above equation holds then it must be that multiplications (1), (2) and (3) are all correct.

3.3 Secure Table Lookup Protocol

We present Protocol 4 that securely computes the ideal functionality $\mathcal{F}_{\mathsf{LUT}}^{\langle\!\langle\cdot\rangle\!\rangle \to [\![\cdot]\!]}$ from Fig. 4. Step 1 executes the ideal functionality $\mathcal{F}_{\mathsf{RandOHV}}$, which performs secure random one-hot vector encoding that takes $N = 2^k$ as input and outputs the shared randomness $[\![r]\!]$ and its corresponding shared random one-hot vector $[\![\boldsymbol{e}^{(r)}]\!]$ (see Fig. 6). Using the randomness, the protocol masks the input and reveals $c = v + r$. The rest of the computation can be done locally as in Step 4-5.

We show that the proposed approach correctly computes the desired value. It is sufficient to show that for each i, the value t_i computed in Step 4 matches the i-th digit $T_v^{(i)}$ of T's v-th element T_v. According to the definition, $t = \bigoplus_{0\leq j\leq n-1} \boldsymbol{e}_j^{(r)} T_{c\oplus j}$, and since the inner product on the right-hand side turns 0 for all terms but $\boldsymbol{e}_r^{(r)} T_{c\oplus r}$, resulting in $t = T_v$.

Functionality $\mathcal{F}_{\mathsf{RandOHV}}$

Input: $N = 2^k$

Output: k random bits $\{[\![r_i]\!]\}_{i=0}^{k-1}$ and length-N one-hot vector $[\![\boldsymbol{e}^{(r)}]\!]$ for $r = r_{k-1}\dots r_0$

(1) $\mathcal{F}_{\mathsf{RandOHV}}$ receives from the adversary $\mathcal{A}$ the shares $\{[\![r_i]\!]^{\mathcal{C}}\}_{i=0}^{k-1}$ and $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{C}}$ considered as the shares of randomness and the corresponding share of the one-hot vector held by corrupted parties.
(2) $\mathcal{F}_{\mathsf{RandOHV}}$ samples $\{r_i\}_{i=0}^{k-1}$ then computes $r = \sum_{i=0}^{k-1} 2^i \cdot r_i$ and $\boldsymbol{e}^{(r)}$.
(3) $\mathcal{F}_{\mathsf{RandOHV}}$ generates $[\![r_i]\!]^{\mathcal{H}}$ from $(r_i, [\![r_i]\!]^{\mathcal{C}})$ for $i \in \{0, 1, \dots, k-1\}$.
(4) $\mathcal{F}_{\mathsf{RandOHV}}$ generates $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ from $\boldsymbol{e}^{(r)}$ and $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{C}}$.
(5) $\mathcal{F}_{\mathsf{RandOHV}}$ distributes the shares $[\![r_i]\!]^{\mathcal{H}}$ and $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ to the honest parties.

Figure 6: The functionality $\mathcal{F}_{\mathsf{RandOHV}}$ to create a random one-hot vector of length N.

Protocol 4 Table lookup of size $N = 2^k$, from $\langle\!\langle\cdot\rangle\!\rangle$ to $[\![\cdot]\!]$ sharing

Functionality: $[\![T_v]\!] \leftarrow \mathcal{F}_{\mathsf{LUT}}(\langle\!\langle v \rangle\!\rangle, T)$
Input: Share $\langle\!\langle v \rangle\!\rangle$ of $v \in GF(2^k)$, table $T : GF(2^k) \longrightarrow GF(2^\ell)$
Output: Share $[\![T_v]\!]$ of the value $T_v \in GF(2^\ell)$
Subfunctionality: $\mathcal{F}_{\mathsf{RandOHV}}$

```
1: $(\{[\![r_i]\!]\}_{0\leq i<k}, \{[\![\boldsymbol{e}_j^{(r)}]\!]\}_{0\leq j<N}) \leftarrow \mathcal{F}_{\mathsf{RandOHV}}(k)$
2: $\langle\!\langle 0 \rangle\!\rangle \leftarrow \mathcal{F}_{\mathsf{Zero}}(GF(2^k))$
3: $c := \mathsf{Reconst}\,(\langle\!\langle v \rangle\!\rangle + \mathsf{ToAdditive}([\![r]\!]) + \langle\!\langle 0 \rangle\!\rangle)$    ▷1 round, 2k bits
4: $[\![t]\!] := \bigoplus_{j=0}^{N-1} [\![\boldsymbol{e}_j^{(r)}]\!] \cdot T_{c\oplus j}$
5: $[\![v]\!] := [\![r]\!] \oplus c$
6: return $([\![t]\!], [\![v]\!])$
```

We prove the following in Appendix D.2.

Lemma 3.2. *The protocol* LUT *in Protocol 4 securely computes $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!] \to \langle\!\langle\cdot\rangle\!\rangle}$ in the $\{\mathcal{F}_{\mathsf{RandOHV}}, \mathcal{F}_{\mathsf{zero}}\}$-hybrid model in the presence of a malicious adversary.*

3.4 General One-Hot Vector Protocol

We now show how to compute the random one-hot vector that was required in the table lookup. We start with a general protocol Ohv to securely compute the one-hot vector of a shared input and turn it to a random one-hot vector protocol by running it on input a set of random shared bits (see Protocol 5). In Appendix C, we also give a specialized protocol for vectors of length 16, with lower round complexity.

We sketch the procedure of Ohv, which is based on the dishonest majority protocol from [40]. It takes $([\![v_{k-1}]\!], \dots, [\![v_0]\!])$ as input. First, it selects bit $[\![v_0]\!]$ and creates a one-hot vector with length 2,

Protocol 5 Random One-hot Vector

```
Functionality: $(\llbracket r \rrbracket, \llbracket \boldsymbol{e}^{(r)} \rrbracket) \leftarrow \mathcal{F}_{\mathsf{RandOHV}}(N = 2^k)$
Input: $\perp$
Output: Shared random bits $\{\llbracket r_i \rrbracket\}_{i=0}^{k-1}$, and length-$N$ one-hot vector
    $\llbracket \boldsymbol{e}^{(r)} \rrbracket$ for $r = \sum_{i=0}^{k-1} 2^i r_i$
1: $(\llbracket r_{k-1} \rrbracket, \dots, \llbracket r_0 \rrbracket) := \mathcal{F}_{\mathsf{rand}}(k)$
2: $\llbracket \boldsymbol{e}^{(r)} \rrbracket := \textsc{Ohv}(\llbracket r_{k-1} \rrbracket, \dots, \llbracket r_0 \rrbracket; N)$
3: Execute $\mathcal{F}_{\mathsf{verify}}$ for the following multiplication triplets from Ohv:
        $(v_{i-1}, f_{2^{i-1}-2}, e_{2^{i-1}-2})$ for all $i \in \{2, \dots, k\}$
4: return $(\{\llbracket r_i \rrbracket\}_{i=0}^{k}, \llbracket \vec{e}^{(r)} \rrbracket)$

5: procedure Ohv($(\llbracket v_{k-1} \rrbracket, \dots, \llbracket v_0 \rrbracket; N = 2^k)$)
6:     if $N = 2$ then
7:         return $(1 - \llbracket v_0 \rrbracket, \llbracket v_0 \rrbracket)$
8:     else
9:         $\llbracket f_0 \rrbracket, \dots, \llbracket f_{N/2-1} \rrbracket := \mathsf{Ohv}(\llbracket v_{k-2} \rrbracket, \dots, \llbracket v_0 \rrbracket; N/2)$
                                            ▷$\log N - 2$ recursive calls with $N > 2$
10:        $\llbracket e_0 \rrbracket, \dots, \llbracket e_{N/2-2} \rrbracket := \mathsf{Mult}(\llbracket v_{k-1} \rrbracket, (\llbracket f_0 \rrbracket, \dots, \llbracket f_{N/2-2} \rrbracket))$
                                            ▷1 round, $N/2 - 1$ bits
11:        $\llbracket e_{N/2-1} \rrbracket := \llbracket v_{k-1} \rrbracket - \bigoplus_{i=0}^{N/2-2} \llbracket e_i \rrbracket$
12:        $\llbracket \boldsymbol{e}^{(v)} \rrbracket := (\llbracket f_0 \rrbracket - \llbracket e_0 \rrbracket, \dots, \llbracket f_{N/2-1} \rrbracket - \llbracket e_{N/2-1} \rrbracket,$
                                            $\llbracket e_0 \rrbracket, \dots, \llbracket e_{N/2-1} \rrbracket)$
                                            ▷$\boldsymbol{e}^{(v)} = ((1 - v_{k-1}) \cdot \vec{f}, v_{k-1} \cdot \vec{f})$
13:        return $\llbracket \boldsymbol{e}^{(v)} \rrbracket$
14:    end if
15: end procedure
```

$(1 - \llbracket v_0 \rrbracket, \llbracket v_0 \rrbracket)$. Then, it selects bit $\llbracket v_1 \rrbracket$ and computes a one-hot vector $((1 - \llbracket v_1 \rrbracket) \cdot (1 - \llbracket v_0 \rrbracket, \llbracket v_0 \rrbracket), \llbracket v_1 \rrbracket \cdot (1 - \llbracket v_0 \rrbracket, \llbracket v_0 \rrbracket))$ with length 4. This is repeated until bit $\llbracket v_{k-1} \rrbracket$ computes a one-hot vector $((1 - \llbracket v_{k-1} \rrbracket) \cdot \vec{f}, \llbracket v_{k-1} \rrbracket \cdot \vec{f})$ with length 2^k, where $\vec{f}$ is a one-hot vector with length 2^{k-1} from the previous iteration.

The communication complexity with three parties is as follows. For general N, the communication cost is $N - \log N - 1$ bits within $\log N - 1$ rounds. For $N = 256$, the communication cost will be 247 bits.

3.5 Approaches Using Large Lookup Tables

In this section, we explore alternative ways of securely computing the AES S-box, using a single lookup table of size 256. We present two protocols for secure AES evaluation with different tradeoffs in communication complexity and round complexity. We also present a third protocol, in Section 3.5.4, which is less efficient for AES, but allows securely evaluating an arbitrary lookup table of size N on $\llbracket \cdot \rrbracket$-shared values, with a communication cost in $O(\sqrt{N})$ instead of $O(N)$. The cost of this protocol (Protocol 10) is shown in Table 3, together with other protocols for comparison.

3.5.1 New $\mathcal{F}_{\mathsf{LUT}}^{ss_1 \mapsto ss_2}$ Instantiations. As building blocks, we use three variants of the $\mathcal{F}_{\mathsf{LUT}}^{ss_1 \mapsto ss_2}$ functionality with different combinations of secret sharing schemes, namely, $\mathcal{F}_{\mathsf{LUT}}^{\llbracket \cdot \rrbracket}$, $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle \cdot \rangle\rangle}$ and $\mathcal{F}_{\mathsf{LUT}}^{\llbracket \cdot \rrbracket \to \langle\langle \cdot \rangle\rangle}$.

In $\mathcal{F}_{\mathsf{LUT}}^{\llbracket \cdot \rrbracket}$, both the input $\llbracket v \rrbracket$ and output $\llbracket T_v \rrbracket$ are given as replicated sharings. This is a stronger requirement than previously, where v was only given additively shared, allowing an adversary to add an error to the input. The simplest way to realize $\mathcal{F}_{\mathsf{LUT}}^{\llbracket \cdot \rrbracket}$ is with a slight tweak to the LUT protocol (Protocol 4): since the input is given in replicated shares, we now run the Reconst procedure (step 3) on the replicated sharing $\llbracket v + r \rrbracket$ instead of $\langle\langle v + r \rangle\rangle$. For a small number of parties, this reduces communication since opening replicated shares is cheaper. For instance, with $n = 3, t = 1$, the cost is reduced from $2k$ bits per party down to just k. Note that the preprocessing cost — generating a random one-hot vector of length $N = 2^k$ via Protocol 5 — is identical to that of $\mathcal{F}_{\mathsf{LUT}}$.

Table 3: Comparison of protocols for table lookup of size $N = 2^k$, with communication complexity for $n = 3, t = 1$. # Mult is the length of the input to $\mathcal{F}_{\mathsf{verify}}$ needed for malicious security

Protocol	Offline	Online	Rounds	# Mult
$\langle\langle \cdot \rangle\rangle \mapsto \llbracket \cdot \rrbracket$ (Prot. 4)	$N - k - 1$	$2k$	1	$k - 1$
$\llbracket \cdot \rrbracket \mapsto \llbracket \cdot \rrbracket$ (Prot. 4 variant)	$N - k - 1$	k	1	$k - 1$
$\langle\langle \cdot \rangle\rangle \mapsto \langle\langle \cdot \rangle\rangle$ (Prot. 6)	$2(\sqrt{N} - \frac{k}{2} - 1)$	$2k$	2	$k - 2$
$\llbracket \cdot \rrbracket \mapsto \langle\langle \cdot \rangle\rangle$ (Prot. 6 variant)	$2(\sqrt{N} - \frac{k}{2} - 1)$	k	1	$k - 2$
$\llbracket \cdot \rrbracket \mapsto \llbracket \cdot \rrbracket$ (Prot. 10)	$2(\sqrt{N} - \frac{k}{2} - 1)$	$2k$	2	N

$\mathcal{F}_{\mathsf{LUT}}^{\langle\langle \cdot \rangle\rangle}$ can be implemented using Protocol 6. This protocol is very similar to that for $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle \cdot \rangle\rangle \to \llbracket \cdot \rrbracket}$ (Protocol 4), except the one-hot vector only needs to be generated in additive shares, rather than replicated shares. This allows for a much more efficient preprocessing protocol: the parties can run the replicated one-hot vector functionality $\mathcal{F}_{\mathsf{RandOHV}}$ twice on input length $2^{k/2}$, obtaining two one-hot vectors $\llbracket \boldsymbol{e}^{(r)} \rrbracket, \llbracket \boldsymbol{e}^{(r')} \rrbracket$. Then, they can locally compute additive shares of the tensor product vector $\boldsymbol{e}^{(r)} \times \boldsymbol{e}^{(r')}$, giving a one-hot vector of length 2^k. Note that, since $\mathcal{F}_{\mathsf{RandOHV}}$ gives replicated shares of the non-zero index, the same shares can still be used to obtain replicated shares of the index of the length 2^k vector.

Protocol 6 Table Lookup of size $N = 2^k$ in additive sharing

```
Functionality: $(\langle\langle T_v \rangle\rangle, \llbracket v \rrbracket) \leftarrow \mathcal{F}_{\mathsf{LUT}}(\langle\langle v \rangle\rangle, T)$
Input: Share $\langle\langle v \rangle\rangle$ of $v \in GF(2^k)$, table $T : GF(2^k) \to GF(2^\ell)$
Output: Share $\langle\langle T_v \rangle\rangle$ of the value $T_v \in GF(2^\ell)$, and share $\llbracket v \rrbracket$
Subfunctionality: $\mathcal{F}_{\mathsf{RandOHV}}$
1: Call $\mathcal{F}_{\mathsf{RandOHV}}(k/2)$ twice to get $(\{\llbracket r_i \rrbracket\}_{i=0}^{k/2-1}, \{\llbracket \boldsymbol{e}_j^{(r)} \rrbracket\}_{j=0}^{\sqrt{N}-1})$
   and $(\{\llbracket r'_i \rrbracket\}_{i=0}^{k/2-1}, \{\llbracket \boldsymbol{e}_j^{(r')} \rrbracket\}_{j=0}^{\sqrt{N}-1})$
2: $\llbracket r \rrbracket := (\llbracket r_0 \rrbracket, \dots, \llbracket r_{k/2-1} \rrbracket, \llbracket r'_0 \rrbracket, \dots, \llbracket r'_{k/2-1} \rrbracket)$
3: $\langle\langle 0 \rangle\rangle \leftarrow \mathcal{F}_{\mathsf{zero}}(GF(2^k))$
4: $c \leftarrow \mathsf{Reconst}\,(\langle\langle v \rangle\rangle + \mathsf{ToAdditive}(\langle\langle r \rangle\rangle) + \langle\langle 0 \rangle\rangle)$    ▷1 round, $2k$ bits
5: $\llbracket v \rrbracket := \llbracket r \rrbracket + c$
6: $\langle\langle \vec{f} \rangle\rangle \leftarrow \left(\mathsf{MultLocal}(\llbracket \boldsymbol{e}_i^{(r)} \rrbracket, \llbracket \boldsymbol{e}_j^{(r')} \rrbracket)\right)_{i,j=0}^{\sqrt{N}-1}$
7: $\langle\langle t \rangle\rangle \leftarrow \bigoplus_{j=0}^{N-1} \langle\langle \vec{f}_j \rangle\rangle \cdot T_{c \oplus j}$
8: return $(\langle\langle t \rangle\rangle, \llbracket v \rrbracket)$
```

The protocol securely realises the functionality $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle \cdot \rangle\rangle}$ from Figure 4. Despite being maliciously secure, one must still take care when composing this protocol with others, since the ideal functionality inherently allows an adversary to cheat by simply changing its additive share of the input or output. In Section 3.5.3, we show how to overcome this issue for the case of AES.

We omit the proof of the following, which is very similar to Lemma 3.2.

LEMMA 3.3. *Protocol 6 securely realizes the functionality $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ in the $(\mathcal{F}_{\mathsf{RandOHV}}, \mathcal{F}_{\mathsf{zero}})$-hybrid model.*

Finally, $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]\to\langle\langle\cdot\rangle\rangle}$ can be implemented using Protocol 6, except the opening of c in step 4 is done on replicated sharings instead of additive. This achieves the strongest performance characteristics of all variants: only k bits of communication per party (for $n = 3, t = 1$) and a cheap preprocessing phase that only requires two replicated, random one-hot vectors of length $2^{k/2}$.

3.5.2 *AES Protocol Based on Replicated Sharing.*

Given the $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ functionality, our protocol for AES evaluation is straightforward. We assume the parties start with replicated shares of the input and expanded key. Then, each S-box is evaluated with a single call to $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ of length $N = 256$, and the linear layers are evaluated locally on the shares. Assuming a maliciously secure implementation of $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$, this protocol is malicious secure, since $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ does not allow any errors to be introduced by the adversary.

With $n = 3, t = 1$, the cost of this protocol in the online phase is just 8 bits of communication per party per S-box, or a total of $10 \cdot 8 \cdot 16 = 1280$ bits for one block of AES. The total round complexity is 10 rounds. The preprocessing phase, however, is much more expensive, due to the need to generate a large, replicated one-hot vector for each S-box. This costs 247 bits per party, per S-box, for a total of 39520 bits. Achieving malicious security can be done by batch verifying each of the multiplications in the RndOhv protocol using $\mathcal{F}_{\mathsf{verify}}$.

3.5.3 *AES Protocol Based on Additive Sharing.*

By relying on $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ instead of $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$, we can reduce the preprocessing cost of the previous protocol by more than 10x. This is because $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ can be realized by generating only two replicated one-hot vectors of length 16, instead of one of length 256. Running the whole protocol on additive instead of replicated shared inputs, we get a 3-party, passively secure protocol with an online communication complexity of 16 bits per S-box (or, 2560 bits overall) and only 22 bits per S-box in the preprocessing phase (or, 3200 overall). The main challenge with this approach is to add malicious security, since we are now dealing with additive shares which are easily tampered with.

We consider two approaches to malicious security. First, we describe a specialized method tailored to AES, which needs no additional communication except for one call to $\mathcal{F}_{\mathsf{verify}}$. The core idea is to exploit the fact that $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ outputs replicated shares of its input x, and use this to obtain replicated shares of the $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ *output* from the previous round, by evaluating the AES linear layer backwards. We combine this with a cheap way of verifying input/output S-box pairs by verifying two multiplication triples, relying on the algebraic structure of the S-box.

Our second approach is more general, and can be used to realize $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ with malicious security for arbitrary lookup tables of domain size 2^k. The preprocessing cost is the same as the AES-specific protocol, but the online phase has slightly more communication, and requires using $\mathcal{F}_{\mathsf{verify}}$ to verify a length-2^k dot product triple, instead of just two multiplications.

Protocol 7 $\langle\langle\cdot\rangle\rangle$-LUT based AES

```
Input: Message $[\![x]\!]$, round keys $\{[\![k^{(i)}]\!], \langle\langle k^{(i)}\rangle\rangle\}_{i=0}^{10}$
Output: $[\![z]\!]$, where $z = AES_k(x)$
 1: $[\![x^0]\!] \leftarrow [\![x]\!] + [\![k^{(0)}]\!]$    ▷AddRoundKey
    ▷$x_b^i, y_b^i$ is byte $b$ of $x^i, y^i$
 2: $\{\langle\langle y_b^0\rangle\rangle\}_{b=0}^{15} \leftarrow \{\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]\to\langle\langle\cdot\rangle\rangle}([\![x_b^0]\!])\}_{b=0}^{15}$    ▷SubBytes
 3: for $i = 1,\ldots,9$ do
 4:     $\langle\langle x^i\rangle\rangle \leftarrow L_i(\langle\langle y^{i-1}\rangle\rangle) + \langle\langle k^{(i)}\rangle\rangle$    ▷Shift/Mix/AddRK
 5:     $\{(\langle\langle y^i\rangle\rangle_b, [\![x_b^i]\!])\}_{b=0}^{15} \leftarrow \{\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}(\langle\langle x_b^i\rangle\rangle)\}_{b=0}^{15}$    ▷SubBytes
 6:     $[\![y^{i-1}]\!] \leftarrow L_i^{-1}([\![x^i]\!] - [\![k^{(i)}]\!])$
 7: end for
 8: $[\![y^9]\!] \leftarrow \mathsf{Reshare}(\langle\langle y^9\rangle\rangle)$
 9: $[\![z]\!] \leftarrow L_{10}([\![y^9]\!]) + [\![k^{(10)}]\!]$    ▷ShiftRows/AddRoundKey
10: triples $\leftarrow \bigcup_{i=0}^{9}\bigcup_{b=0}^{15} \textsc{VerifySbox}([\![x_b^i]\!], \mathsf{Affine}^{-1}([\![y_b^i]\!]))$
11: Run $\mathcal{F}_{\mathsf{verify}}$ to check triples
12: return $[\![z]\!]$

13: procedure $\textsc{VerifySbox}([\![x]\!], [\![y]\!])$
14:     $[\![x^2]\!] = \mathsf{Square}([\![x]\!])$
15:     $[\![y^2]\!] = \mathsf{Square}([\![y]\!])$
16:     return $\{([\![x^2]\!], [\![y]\!], [\![x]\!]), ([\![x]\!], [\![y^2]\!], [\![y]\!])\}$
17: end procedure
```

AES-Optimized Protocol. We present the full protocol for AES evaluation in Protocol 7. The protocol begins with the inputs and round keys distributed as replicated shares. The first round of S-boxes is computed with $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]\to\langle\langle\cdot\rangle\rangle}$. For each subsequent round, we proceed to evaluate the linear layer, denoted L_i, and round key addition, followed by the S-box with $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ to get an S-box output $y_i = \mathrm{SubBytes}(x_i)$. We then invert the linear layer on the replicated shares of x_i to recover replicated shares of y_{i-1}. Finally, each S-box input/output pair (x_i, y_i) is verified with $\mathcal{F}_{\mathsf{verify}}$, by first inverting the affine component of the S-box, denoted Affine, to get $\hat{y}_i$, and checking the two equations: $x_i^2\hat{y}_i = x_i$, and $x_i\hat{y}_i^2 = \hat{y}_i$ which hold if and only if $\hat{y}_i = x_i^{254}$ in $GF(2^8)$. This idea was recently proposed for zero-knowledge proofs of AES [8].

One subtlety of the security proof (in Appendix D.3) is that when using $\mathcal{F}_{\mathsf{verify}}$, we *cannot* allow the adversary to learn the errors in the multiplication triples, e.g. the values $d_i = x_i^2\hat{y}_i - x_i$. This is because an error in an S-box input corresponds to an error in x_i, which would lead to a non-zero value of d_i that leaks information on $\hat{y}_i$. While at first glance, it may seem that even the *presence* of input-dependent errors would leak information to the adversary, in our case it is not a security issue to reveal whether *some* error occurred: if any error d_i is non-zero then at least one of $x_i^2\hat{y}_i = x_i$ or $x_i\hat{y}_i^2 = \hat{y}_i$ must be false. The key point is that we cannot reveal the value or location of this error, which is why we need the stronger $\mathcal{F}_{\mathsf{verify}}$ functionality from Section 2.5.

We prove the following in Appendix D.3.

Lemma 3.4. *Protocol 7 securely realizes the functionality $\mathcal{F}_{\mathsf{AES}}$ in the $(\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}, \mathcal{F}_{\mathsf{verify}})$-hybrid model with malicious security.*

3.5.4 Improved Protocol for $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$. In Protocol 10, shown in Appendix E, we present an alternative protocol for $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$, which generalizes the ideas of the previous protocol to arbitrary lookup tables. Compared with the naive protocol for $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ discussed in the previous section, we reduce the communication cost of the preprocessing to $O(\sqrt{N})$. The online phase, however, has roughly double the cost in terms of communication and rounds. This is not as efficient as the AES-specific protocol in the previous section, but may still be useful in other applications.

The protocol follows a similar approach to Protocol 6, building a length-N one-hot vector taking the tensor product of two length-$\sqrt{N}$ vectors, except it works on replicated shared inputs. Recall that Protocol 6 computes the output $v = \bigoplus_j \vec{f}_j \cdot T_{c\oplus j}$, where c is a masked version of the input and $\vec{f}$ is the one-hot vector. We observe that, when $\vec{f}$ is decomposed into a tensor product of two smaller one-hot vectors, v can be seen as an inner product of two *secret*, $[\![\cdot]\!]$-shared vectors of length N. This means we can tweak this to obtain *replicated* shares of the output, by using the $\mathcal{F}_{\mathsf{weakDotProduct}}$ functionality, followed by $\mathcal{F}_{\mathsf{verify}}$ to obtain malicious security.

4 PERFORMANCE

We implemented several variants of our protocol and two most related protocols from the state of the art in the same software framework. This ensures a fair comparison and minimizes performance differences due to different networking architectures, primitives used for PRNGs and hashing, etc. The two state-of-the-art protocols serve as a baseline for the comparison. In the semi-honest security setting, we use Chida et al.'s $GF(2^8)$-Circuit protocol as described in [21]. In the malicious security setting, we adapt $GF(2^8)$-Circuit using bucket cut-and-choose to generate correct multiplication triples in the offline phase that are used to verify the correctness of the multiplications in the online phase using sacrificing. The techniques are adapted from [31] but in $GF(2^8)$.

Below, we describe details of the implementation and include notable optimizations that were applied. This is followed by a discussion of the experimental setup and the performance benchmark.

4.1 Implementation

We implemented the protocols from scratch using the Rust programming language. Our code is available[2]. Non-linear operations of small fields, e.g., $GF(2^8)$, $GF(2^4)$, are implemented via table lookups. Networking and I/O is done in a separate thread where each channel between two parties is encrypted and mutually authenticated using TLS1.3 with client/server certificates. Local randomness, for instance to implement $\mathcal{F}_{\mathsf{rand}}$, comes from a PRNG based on Chacha20, the hash function we use for compare-view is SHA-256. The data representation is mostly in the struct of arrays style to allow for SIMD auto-vectorization by the compiler and I/O with lower memory copying. Next, we describe the implemented protocols in detail.

[2]in the public repository https://github.com/KULeuven-COSIC/maestro.

$GF(2^8)$**-Circuit** Baseline for semi-honest security, described in [21] in Algorithm 5.

Mal. $GF(2^8)$**-Circuit** Baseline for malicious security. In the offline phase, multiplication triples are generated using bucket cut-and-choose with bucket size $B = 3$ and $C = 3$ triples to open described in [31] (for $\geq 2^{19}$ multiplications, i.e., ≥ 820 AES blocks). After computing the circuit but before revealing the output, the multiplications from the online phase are checked by sacrificing the preprocessed triples. We call this phase post-sacrifice.

LUT-16 This variant implements Protocol 3 using Protocol 9 in the offline phase to generate random one-hot vectors. For malicious security, Protocol 2 is used to verify correctness of multiplications.

$GF(2^4)$**-Circuit** This variant implements Protocol 3 but computes the inverse $[\![v^{-1}]\!]$ as $[\![v^2]\!] \cdot [\![v^4]\!] \cdot [\![v^8]\!]$ which does not require preprocessing. For malicious security, Protocol 2 is used to verify correctness of multiplications.

(2,3) LUT-256 In this variant, the LUT protocol (see Protocol 4) computes the complete AES S-box consuming one random one-hot vector of size 256 from the offline phase. The offline phase creating the one-hot vectors is implemented with Protocol 5. We did not implement the maliciously secure variant.

For LUT-16 and LUT-256, the offline phase computes on bit shares. We implemented the generation of random one-hot vectors using bit-slicing where we operate on a pack of 16 bits. This improves both local computation and efficiency for I/O. The inner product required in oblivious table lookups is realized using 16 and 256 hard-coded tables, respectively, that contain the lookup table permuted by the reconstructed public $c \oplus j$ (see Step 4 in Protocol 4). Specifically, the table entry contains 4 and 8 bitvectors, respectively, where each encodes the i-th output bit of the permuted table. Then, given the random one-hot vector $\vec{e}$ as bitvector, the inner product for each output bit can be computed as $(\vec{e}$ `&` $\vec{t[i]})$`.parity() mod 2` where & denotes bit-wise AND. This approach neither requires branching nor multiplication instructions and improves local computation of Protocol 4 by about 10 times compared to a naive approach.

Protocol 2 is implemented in $GF(2^{64})$ to achieve an acceptable level of soundness of at least 40 bits and utilize hardware support for carry-less multiplication (CLMUL). We also gain some efficiency by doing modular reduction only once at the end when computing inner products. Protocol 7 is not implemented.

4.2 Experimental Setup

We experimentally evaluate the performance of the proposed protocols in two different setups and settings. In the high throughput setting, we evaluate on three machines with identical specifications (16-core Intel Core i9-9900 3.10GHz, 128GB RAM) connected via a network with $\approx$ 9.4 Gbits/sec bandwidth and <1ms latency. To observe the protocol behaviour in different network settings, we evaluate on three machines (each 16-core Intel XEON E5-2650v2 2.60GHz, 128 GB RAM) connected via a network of $\approx$ 950 MBits/sec, <1ms latency and $\approx$ 50 Mbits/sec with 100ms round

Table 4: Benchmark results for passive security on batches of 250 000 AES blocks in the LAN setting with ≈ 9.42 Gbits/sec bandwidth. Time and communicated data is reported per batch, the throughput is reported as AES blocks per second. We denote the best value for online phase and throughput in bold.

Protocol	Preprocessing		Online		Throughput (blocks/s)		
	Time (s)	Data (MB)	Time (s)	Data (MB)	Preprocessing	Online	Total
$GF(2^8)$-Circuit [21]	-	-	0.410	160	-	610 701	610 701
LUT-16	0.505	55	**0.321**	80	494 712	**777 943**	302 405
$GF(2^4)$-Circuit	-	-	0.331	100		755 621	**755 621**
(2,3) LUT-256	9.53	1235	0.663	**40**	26 223	377 282	24 519

Table 5: Benchmark results for active security on batches of 100 000 AES blocks in the LAN setting with ≈ 9.42 Gbits/sec bandwidth. Time and communicated data is reported per batch, the throughput is reported as AES blocks per second. We denote the best value per metric in bold.

Protocol	Preprocessing		Online		Throughput (blocks/s)		
	Time (s)	Data (MB)	Time (s)	Data (MB)	Preprocessing	Online	Total
Mal. $GF(2^8)$-Circuit ([21]+[31])	9.47	≈ 470	**0.655**	≈ 192	10 556	**152 744**	9 874
LUT-16	**0.242**	**22**	6.68	≈ 40	413 661	14 972	14 449
LUT-16 (prep)	6.61	≈ 22	1.86	$\approx$ **40**	15 137	53 880	11 817
LUT-16 (prep+sbox)	6.61	≈ 22	2.28	≈ 40	15 100	43 853	11 232
$GF(2^4)$-Circuit	-	-	3.61	≈ 40	-	27 686	27 686
$GF(2^4)$-Circuit (sbox)	-	-	2.73	≈ 40	-	36 585	**36 585**

trip time. The network throughput/latency was altered using `tc`. In both settings, 16 computation threads were used.

We measure execution time (wall clock time) and the total amount of bytes sent per party during the computation of a AES block enciphering without the keyschedule. To amortize performance, we compute on S many AES blocks in parallel. The time/communication for preprocessing and online phase includes all necessary checks for malicious security (e.g., Protocol 2 and compare-view). We don't measure the negligible setup phase for the correlated randomness.

The data that is reported in the following benchmark is the execution time/data communication of the slowest party, averaged over at least 10 iterations of the protocol. The throughput (denoted in AES blocks per second) is computed as $\lfloor S/t \rfloor$ where S are the number of AES blocks and t is the execution time in seconds.

4.3 Benchmark and Discussion

We first report on the high-throughput setting for semi-honest security in Table 4 and malicious security in Table 5. For semi-honest security, two of our protocols, LUT-16 and $GF(2^4)$-Circuit outperform the state-of-the-art $GF(2^8)$-Circuit protocol. LUT-16 offers the fastest online phase which improves online throughput by 27% compared to $GF(2^8)$-Circuit, while $GF(2^4)$-Circuit has the highest, overall throughput resulting in a $\approx 23\%$ improvement compared to $GF(2^8)$-Circuit. The $(2,3)$ LUT-256 protocol variant allows for a potentially rapid online phase due to the few communication rounds and low amount of data. Our current implementation cannot fully realize this potential. The bottleneck is the local computation of the inner product between the random one-hot bitvector and the permuted 256-element lookup table. Further optimization is required for this step. This poor performance coupled with the high cost of the preprocessing makes the variant not attractive for the malicious security setting, so we didn't implement it.

For malicious security, we first note that our implementation of the multiplication correctness check of Protocol 2 is comparatively much slower than, e.g., the triple post-sacrifice step in [31], despite optimizations using carry-less multiplication for $GF(2^{64})$ multiplication and inner products (about 3 to 4 times). This results in a significantly slower online phase for our protocols. Due to this observation, we included variants that compute an intermediate verification step after preprocessing (prep) and after every S-box layer (sbox). However, regarding overall throughput, both LUT-16 and $GF(2^4)$-Circuit outperform the maliciously secure $GF(2^8)$-Circuit protocol with improvements ranging from 46% and 270%, respectively. It is also noteworthy that our protocol variants use much less communication, decreasing the number of sent bytes by 90% (LUT-16) and 93% ($GF(2^4)$-Circuit).

We now report on the protocol performance for varying network settings in Table 6. In the setting of semi-honest security, $GF(2^4)$-Circuit improves overall throughput by about 37% in the 1 Gbit/s LAN, and by 72% in the WAN setting, while LUT-16 also improves online phase throughput by 33% and 71%. The high latency of the WAN setting even allows (2,3) LUT-256 to improve the online throughput by 16%, however total throughput is still less than all other protocols due to the expensive preprocessing.

For active security, we make two observations. First, our protocols consistently increase total throughput in both 1 Gbit/s LAN

Table 6: The throughput in AES blocks per second for different network settings. We denote the best value for online and total throughput in bold.

Network	Protocol	Throughput (blocks/s)		
		Preprocessing	Online	Total
	$GF(2^8)$-Circuit [21]	-	610701	610701
	LUT-16	494712	**777943**	302405
	$GF(2^4)$-Circuit	-	755621	**755621**
10 Gbit/s	$(2,3)$ LUT-256	26223	377282	24519
≤ 1ms RTT	Mal. $GF(2^8)$-Circuit ([21]+[31])	10556	**152744**	9874
	(Mal.) LUT-16	413661	14972	14449
	(Mal.) LUT-16 (prep)	15137	53880	11817
	(Mal.) $GF(2^4)$-Circuit (sbox)	-	36585	**36585**
	$GF(2^8)$-Circuit	-	118551	118551
	LUT-16	201719	158333	88706
	$GF(2^4)$-Circuit	-	**162949**	**162949**
1 Gbit/s	$(2,3)$ LUT 256	9208	89508	8349
≤ 1ms RTT	Mal. $GF(2^8)$-Circuit	2781	**34073**	2571
	(Mal.) LUT-16	155580	3877	3783
	(Mal.) LUT-16 (prep)	7676	24376	5838
	(Mal.) $GF(2^4)$-Circuit	-	14227	**14227**
	(Mal.) $GF(2^4)$-Circuit (sbox)	-	11388	11388
	$GF(2^8)$-Circuit	-	6341	6341
	LUT-16	16188	10903	6515
	$GF(2^4)$-Circuit	-	**10932**	**10932**
50 Mbit/s	$(2,3)$ LUT 256	1111	7371	965
100ms RTT	Mal. $GF(2^8)$-Circuit	1002	2843	741
	(Mal.) LUT-16	14655	3489	2818
	(Mal.) LUT-16 (prep)	4138	**6275**	2493
	(Mal.) $GF(2^4)$-Circuit	-	4877	**4877**

and WAN with improvements ranging from 127% and 453% for LUT-16 and $GF(2^4)$-Circuit, respectively. Second, unlike in the 10 Gbit/s network, now in the WAN setting, the post-sacrifice employed in Mal. $GF(2^8)$-Circuit sends too much data and thus our multiplication check enables improvements in the online phase from 120% and 71%, respectively for LUT-16 and $GF(2^4)$-Circuit.

While our protocol implementation was geared towards high-throughput, it is possible to get an approximation of the computation latency by evaluating only one AES block (see Table 7). In the WAN setting, lookup-table based approaches with a lower number of rounds therefore have a lower latency. LUT-16 and LUT-256 reduce latency by 26% and 65% compared to $GF(2^8)$-Circuit, respectively. Our $GF(2^4)$-Circuit variant requires 4 rounds per S-box and thus increases computation latency by 11%.

Chida et al. [21] also report on another setup where machines are connected in a ring topology with dual connections between each machine. This allows for optimizations where for each step, half of the data is sent to one party, and half is sent to the other party, essentially rotating the parties' roles to improve throughput. Moreover, they also implement counter-mode caching as a mode-level optimization. Both optimizations can also be applied to our protocols and should also benefit our protocols.

ACKNOWLEDGMENTS

This work was partly supported by the Flemish Government through FWO SBO project MOZAIK S003321N, by CyberSecurity Research Flanders with reference number VR20192203, by the Danish Independent Research Council under Grant-ID DFF-0165-00107B (C3PO), by the Digital Research Center Denmark (DIREC), by the DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics, by JSPS KAKENHI Grant Number JP21H05052, and by JST CREST Grant Number JPMJCR22M1.

Table 7: Computation latency for one AES block, reported as the execution time of the online phase in the WAN setting.

Protocol	Latency (in ms)
$GF(2^8)$-Circuit [21]	2935
LUT-16	2169
$GF(2^4)$-Circuit	3279
LUT-256	**1026**
Mal. $GF(2^8)$-Circuit ([21]+[31])	2866
(Mal.) LUT-16 (prep)	3521
(Mal.) $GF(2^4)$-Circuit	4577

REFERENCES

[1] Aysajan Abidin, Enzo Marquet, Jerico Moeyersons, Xhulio Limani, Erik Pohle, Michiel Van Kenhove, Johann M. Márquez-Barja, Nina Slamnik-Krijestorac, and Bruno Volckaert. 2023. MOZAIK: An End-to-End Secure Data Sharing Platform. In *Proceedings of the Second ACM Data Economy Workshop, DEC 2023, Seattle, WA, USA, 18 June 2023*. ACM, 34–40. https://doi.org/10.1145/3600046.3600052

[2] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. 2021. Oblivious TLS via Multi-party Computation. In *CT-RSA 2021 (LNCS, Vol. 12704)*, Kenneth G. Paterson (Ed.). Springer, Heidelberg, 51–74. https://doi.org/10.1007/978-3-030-75539-3_3

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

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

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. 2017. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 843–862. https://doi.org/10.1109/SP.2017.15

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 805–817. https://doi.org/10.1145/2976749.2978331

[7] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, and Kazunari Tozawa. 2022. Memory and Round-Efficient MPC Primitives in the Pre-Processing Model from Unit Vectorization. In *ASIACCS 22*, Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako (Eds.). ACM Press, 858–872. https://doi.org/10.1145/3488932.3517407

[8] Carsten Baum, Ward Beullens, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. 2024. One Tree to Rule Them All: Optimizing GGM Trees and OWFs for Post-Quantum Signatures. Cryptology ePrint Archive, Paper 2024/490. https://eprint.iacr.org/2024/490 https://eprint.iacr.org/2024/490.

[9] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. 2020. PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 587–606. https://doi.org/10.1109/EUROSP48549.2020.00044

[10] Amit Singh Bhati, Erik Pohle, Aysajan Abidin, Elena Andreeva, and Bart Preneel. 2023. Let's Go Eevee! A Friendly and Suitable Family of AEAD Modes for IoT-to-Cloud Secure Computation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS*

2023, Copenhagen, Denmark, November 26-30, 2023, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2546–2560. https://doi.org/10.1145/3576915.3623091

[11] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO 2019, Part III (LNCS, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Heidelberg, 67–97. https://doi.org/10.1007/978-3-030-26954-8_3

[12] Joan Boyar, Philip Matthews, and René Peralta. 2013. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology* 26, 2 (April 2013), 280–312. https://doi.org/10.1007/s00145-012-9124-7

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 1292–1303. https://doi.org/10.1145/2976749.2978429

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *TCC 2019, Part I (LNCS, Vol. 11891)*, Dennis Hofheinz and Alon Rosen (Eds.). Springer, Heidelberg, 341–371. https://doi.org/10.1007/978-3-030-36030-6_14

[15] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 869–886. https://doi.org/10.1145/3319535.3363227

[16] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2020. Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs. In *ASIACRYPT 2020, Part III (LNCS, Vol. 12493)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 244–276. https://doi.org/10.1007/978-3-030-64840-4_9

[17] Luís T. A. N. Brandão, Nicolas Christin, George Danezis, and Anonymous. 2015. Toward Mending Two Nation-Scale Brokered Identification Systems. *PoPETs* 2015, 2 (April 2015), 135–155. https://doi.org/10.1515/popets-2015-0022

[18] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame. 2023. FLUTE: Fast and Secure Lookup Table Evaluations. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 515–533. https://doi.org/10.1109/SP46215.2023.00157

[19] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[20] D. Canright. 2005. A Very Compact S-Box for AES. In *CHES 2005 (LNCS, Vol. 3659)*, Josyula R. Rao and Berk Sunar (Eds.). Springer, Heidelberg, 441–455. https://doi.org/10.1007/11545262_32

[21] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Benny Pinkas. 2018. High-Throughput Secure AES Computation. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018, Toronto, ON, Canada, October 19, 2018*, Michael Brenner and Kurt Rohloff (Eds.). ACM, 13–24. https://doi.org/10.1145/3267973.3267977

[22] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC 2005 (LNCS, Vol. 3378)*, Joe Kilian (Ed.). Springer, Heidelberg, 342–362. https://doi.org/10.1007/978-3-540-30576-7_19

[23] Ivan Damgård and Marcel Keller. 2010. Secure Multiparty AES. In *FC 2010 (LNCS, Vol. 6052)*, Radu Sion (Ed.). Springer, Heidelberg, 367–374.

[24] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. 2012. Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol. In *SCN 12 (LNCS, Vol. 7485)*, Ivan Visconti and Roberto De Prisco (Eds.). Springer, Heidelberg, 241–263. https://doi.org/10.1007/978-3-642-32928-9_14

[25] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2017. The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited. In *CRYPTO 2017, Part I (LNCS, Vol. 10401)*, Jonathan Katz and Hovav Shacham (Eds.). Springer, Heidelberg, 167–187. https://doi.org/10.1007/978-3-319-63688-7_6

[26] Ivan Damgård and Rasmus Winther Zakarias. 2016. Fast Oblivious AES A Dedicated Application of the MiniMac Protocol. In *AFRICACRYPT 16 (LNCS, Vol. 9646)*, David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer, Heidelberg, 245–264. https://doi.org/10.1007/978-3-319-31517-1_13

[27] F. Betül Durak and Jorge Guajardo. 2021. Improving the Efficiency of AES Protocols in Multi-Party Computation. In *FC 2021, Part I (LNCS, Vol. 12674)*, Nikita Borisov and Claudia Díaz (Eds.). Springer, Heidelberg, 229–248. https://doi.org/10.1007/978-3-662-64322-8_11

[28] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. 2001. Advanced Encryption Standard (AES). https://doi.org/10.6028/NIST.FIPS.197

[29] J.L. Fan and C. Paar. 1997. On efficient inversion in tower fields of characteristic two. In *Proceedings of IEEE International Symposium on Information Theory*. 20–. https://doi.org/10.1109/ISIT.1997.612935

[30] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3378)*, Joe Kilian (Ed.). Springer, 303–324. https://doi.org/10.1007/978-3-540-30576-7_17

[31] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT 2017, Part II (LNCS, Vol. 10211)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, 225–255. https://doi.org/10.1007/978-3-319-56614-6_8

[32] Vipul Goyal and Yifan Song. 2020. Malicious Security Comes Free in Honest-Majority MPC. Cryptology ePrint Archive, Report 2020/134. https://eprint.iacr.org/2020/134.

[33] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *CRYPTO 2020, Part II (LNCS, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 618–646. https://doi.org/10.1007/978-3-030-56880-1_22

[34] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. 2016. MPC-Friendly Symmetric Key Primitives. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 430–443. https://doi.org/10.1145/2976749.2978332

[35] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. 2015. Fast Garbling of Circuits Under Standard Assumptions. In *ACM CCS 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, 567–578. https://doi.org/10.1145/2810103.2813619

[36] Carmit Hazay and Yehuda Lindell. 2008. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In *TCC 2008 (LNCS, Vol. 4948)*, Ran Canetti (Ed.). Springer, Heidelberg, 155–175. https://doi.org/10.1007/978-3-540-78524-8_10

[37] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security 2011*. USENIX Association.

[38] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.

[39] Toshiya Itoh and Shigeo Tsujii. 1988. A Fast Algorithm for Computing Multiplicative Inverses in GF(2m) Using Normal Bases. *Inf. Comput.* 78, 3 (sep 1988), 171–177. https://doi.org/10.1016/0890-5401(88)90024-7

[40] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. 2017. Faster Secure Multi-party Computation of AES and DES Using Lookup Tables. In *ACNS 17 (LNCS, Vol. 10355)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.). Springer, Heidelberg, 229–249. https://doi.org/10.1007/978-3-319-61204-1_12

[41] Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood. 2018. Reducing Communication Channels in MPC. In *SCN 18 (LNCS, Vol. 11035)*, Dario Catalano and Roberto De Prisco (Eds.). Springer, Heidelberg, 181–199. https://doi.org/10.1007/978-3-319-98113-0_10

[42] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. 2012. Efficient Lookup-Table Protocol in Secure Multiparty Computation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2364527.2364556

[43] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *ACNS 13 (LNCS, Vol. 7954)*, Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer, Heidelberg, 84–101. https://doi.org/10.1007/978-3-642-38980-1_6

[44] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM CCS 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, 591–602. https://doi.org/10.1145/2810103.2813705

[45] Sean Murphy and Matthew J. B. Robshaw. 2002. Essential Algebraic Structure within the AES. In *CRYPTO 2002 (LNCS, Vol. 2442)*, Moti Yung (Ed.). Springer, Heidelberg, 1–16. https://doi.org/10.1007/3-540-45708-9_1

[46] Peter Sebastian Nordholt and Meilof Veeningen. 2018. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *ACNS 18 (LNCS, Vol. 10892)*, Bart Preneel and Frederik Vercauteren (Eds.). Springer, Heidelberg, 321–339. https://doi.org/10.1007/978-3-319-93387-0_17

[47] Satsuya Ohata and Koji Nuida. 2020. Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application. In *FC 2020 (LNCS, Vol. 12059)*, Joseph Bonneau and Nadia Heninger (Eds.). Springer, Heidelberg, 369–385. https://doi.org/10.1007/978-3-030-51280-4_20

[48] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *ASIACRYPT 2009 (LNCS, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, Heidelberg, 250–267. https://doi.org/10.1007/978-3-642-10366-7_15

[49] Mike Rosulek and Lawrence Roy. 2021. Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits. In *CRYPTO 2021, Part I (LNCS, Vol. 12825)*, Tal Malkin and Chris Peikert (Eds.). Springer, Heidelberg, Virtual Event, 94–124. https://doi.org/10.1007/978-3-030-84242-0_5

[50] Dragos Rotaru, Nigel P. Smart, and Martijn Stam. 2017. Modes of Operation Suitable for Computing on Encrypted Data. *IACR Trans. Symm. Cryptol.* 2017, 3 (2017), 294–324. https://doi.org/10.13154/tosc.v2017.i3.294-324

[51] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. 2001. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *ASIACRYPT 2001 (LNCS, Vol. 2248)*, Colin Boyd (Ed.). Springer, Heidelberg, 239–254. https://doi.org/10.1007/3-540-45682-1_15

[52] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. 2002. An ASIC Implementation of the AES S-Boxes. In *CT-RSA 2002 (LNCS, Vol. 2271)*, Bart Preneel (Ed.). Springer, Heidelberg, 67–78. https://doi.org/10.1007/3-540-45760-7_6

[53] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT 2015, Part II (LNCS, Vol. 9057)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Heidelberg, 220–250. https://doi.org/10.1007/978-3-662-46803-6_8

A DETAILS ON AES AND FIELD ISOMORPHISM

A.1 AES

A.1.1 Encryption Algorithm for Each Block. The encryption of a block in AES is a deterministic algorithm that takes a 128-bit array and an encryption key as input and produces a 128-bit array as output. The algorithm can be defined as a function $\mathsf{Enc} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$.

The protocol proceeds as follows:

(1) **Initialization:** The input values are divided into 8-bit segments, each of which is considered an element of $GF(2^8)$. These elements are represented in a 4×4 array with column-first order, denoted as $\{s_{r,c}\}_{0\le r,c\le 3}$. The same process is applied to the encryption key.

(2) **Key Expansion:** The round keys $\{k^{(i)}_{r,c}\}_{0\le r,c\le 3, 0\le i\le 10}$ to be used in each round $i \in \{1,\dots,10\}$ are generated from the encryption keys $\{k_{r,c}\}_{0\le r,c\le 3}$. This is done as follows:

$$k^{(i)}_{r,c} := \begin{cases} k_{r,c} & \text{if } i = 0\,, \\ k^{(i-1)}_{r,0} \oplus \mathsf{Sbox}(k^{(i-1)}_{(r+1 \bmod 4),3}) \oplus rc^{(i)}_r & \text{if } i \neq 0, c = 0\,, \\ k^{(i-1)}_{r,c} \oplus k^{(i)}_{r,c-1} & \text{otherwise}\,. \end{cases}$$

Here, $rc^{(i)}_r \in GF(2^8)$ is defined as $rc^{(i)}_0 := (\{02\}_{16})^{i-1}$ and $rc^{(i)}_r := \{00\}_{16}$ for $1 \le r \le 3$. Sbox represents a substitution according to a predefined table (Section A.1.2).

(3) Each element of the array is computed as $s_{r,c} := s_{r,c} \oplus k^{(0)}_{r,c}$.

(4) **Round Processing:** For $i = 1,\dots,10$, the following steps are repeated:

- **SubBytes:** Each element of the array is substituted according to a predefined table (AES S-box):

$$s_{r,c} := \mathsf{Sbox}(s_{r,c})$$

- **ShiftRows:** Each row is shifted according to the following rule:

$$s_{r,c} := s_{r,(c+r \bmod 4)}$$

- **MixColumns:** For each column, the following are calculated:

$$\begin{pmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{pmatrix} := \begin{pmatrix} \{02\}_{16} & \{03\}_{16} & \{01\}_{16} & \{01\}_{16} \\ \{01\}_{16} & \{02\}_{16} & \{03\}_{16} & \{01\}_{16} \\ \{01\}_{16} & \{01\}_{16} & \{02\}_{16} & \{03\}_{16} \\ \{03\}_{16} & \{01\}_{16} & \{01\}_{16} & \{02\}_{16} \end{pmatrix} \begin{pmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{pmatrix}$$

Note that this step is omitted when $i = 10$.

- **AddRoundKey:** Each element of the array is XORed with the round key:

$$s_{r,c} := s_{r,c} \oplus k^{(i)}_{r,c}$$

(5) **Finalization:** The 4×4 array $\{s_{r,c}\}$ is concatenated in column-first order to produce the output.

A.1.2 AES S-Box. The AES S-Box is a substitution table used in the key expansion and SubBytes step to ensure the non-linearity of encryption. Specifically, for an input $s \in GF(2^8)$, it produces an output $\{a_7 \cdots a_0\}_2 \in GF(2^8)$ defined as follows:

$$\begin{aligned} a_0 &:= b_0 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus 1 \\ a_1 &:= b_0 \oplus b_1 \oplus b_5 \oplus b_6 \oplus b_7 \oplus 1 \\ a_2 &:= b_0 \oplus b_1 \oplus b_2 \oplus b_6 \oplus b_7 \\ a_3 &:= b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_7 \\ a_4 &:= b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \\ a_5 &:= b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus 1 \\ a_6 &:= b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus 1 \\ a_7 &:= b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 . \end{aligned} \tag{3}$$

Here, $\{b_7 \cdots b_0\}_2$ represents the bit sequence that denotes the *multiplicative inverse* $s^{-1} \in GF(2^8)$ of the input value $s \in GF(2^8)$. Note that when $s = \{00\}_{16}$, all b_i are set to 0 for all i.

A.1.3 Oblivious AES. Protocol 8 describes the whole AES algorithm that is computed in MPC. The main body of the paper focused on computing SubBytes.

A.2 Finite Field Isomorphism

Below, we give the explicit isomorphism between $GF(2^8)$ and $GF((2^4)^2)$, from [52]:

$$\Phi\colon GF(2^8) \xrightarrow{\sim} GF((2^4)^2) : \{a_7 a_6 \dots a_0\}_2 \longmapsto (a_h, a_\ell)$$

$$\begin{aligned} a_{h0} &:= a_4 \oplus a_5 \oplus a_6, & a_{\ell 0} &:= a_0 \oplus a_4 \oplus a_5 \oplus a_6 \\ a_{h1} &:= a_1 \oplus a_4 \oplus a_6 \oplus a_7, & a_{\ell 1} &:= a_1 \oplus a_2 \\ a_{h2} &:= a_2 \oplus a_3 \oplus a_5 \oplus a_7, & a_{\ell 2} &:= a_1 \oplus a_7 \\ a_{h3} &:= a_5 \oplus a_7, & a_{\ell 3} &:= a_2 \oplus a_4 \end{aligned}$$

The inverse mapping is defined as follows:

$$\Phi^{-1}\colon GF((2^4)^2) \xrightarrow{\sim} GF(2^8) : (a_h, a_\ell) \longmapsto \{a_7 a_6 \dots a_0\}_2$$

$$\begin{aligned} a_0 &:= a_{\ell 0} \oplus a_{h0}, & a_4 &:= a_{\ell 1} \oplus a_{\ell 3} \oplus a_{h0} \oplus a_{h1} \oplus a_{h3} \\ a_1 &:= a_{h0} \oplus a_{h1} \oplus a_{h3}, & a_5 &:= a_{\ell 2} \oplus a_{h0} \oplus a_{h1} \\ a_2 &:= a_{\ell 1} \oplus a_{h0} \oplus a_{h1} \oplus a_{h3}, & a_6 &:= a_{\ell 1} \oplus a_{\ell 2} \oplus a_{\ell 3} \oplus a_{h0} \oplus a_{h3} \\ a_3 &:= a_{h0} \oplus a_{h1} \oplus a_{h2} \oplus a_{\ell 1}, & a_7 &:= a_{\ell 2} \oplus a_{h0} \oplus a_{h1} \oplus a_{h3} \end{aligned}$$

Protocol 8 Oblivious AES

Functionality: $\{[\![z_i]\!]\} \leftarrow \mathcal{F}_{\mathsf{AES}}(\{[\![x_i]\!]\}, \{[\![k_i]\!]\})$

Input: 128-bit Boolean shared values $\{[\![x_i]\!]\}_{i=0,\dots,127}$, $\{[\![k_i]\!]\}_{i=0,\dots,127}$

Output: 128-bit Boolean shared value $\{[\![z_i]\!]\}_{i=0,\dots,127}$

Subfunctionality:

```
1: Servers locally perform initialization step to obtain $\{[\![x_{r,c}]\!]\}_{0\le r,c\le 3}$ and $\{[\![k_{r,c}]\!]\}_{0\le r,c\le 3}$
2: $\{[\![k^{(i)}_{r,c}]\!]\}_{0\le i\le 10, 0\le r,c\le 3} \leftarrow \mathsf{KeyExpansion}(\{[\![k_{r,c}]\!]\}_{0\le r,c\le 3})$
                                                  ▹one-time operation for each key
3: $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{AddRoundKey}(\{[\![x_{r,c}]\!]\}, \{[\![k^{(0)}_{r,c}]\!]\})$
4: for $i = 1, \dots, 9$ do
5:     $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{SubBytes}(\{[\![x_{r,c}]\!]\})$
6:     $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{ShiftRows}(\{[\![x_{r,c}]\!]\})$
7:     $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{MixColumns}(\{[\![x_{r,c}]\!]\})$
8:     $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{AddRoundKey}(\{[\![x_{r,c}]\!]\}, \{[\![k^{(i)}_{r,c}]\!]\})$
9: end for
10: $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{SubBytes}(\{[\![x_{r,c}]\!]\})$
11: $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{ShiftRows}(\{[\![x_{r,c}]\!]\})$
12: $\{[\![x_{r,c}]\!]\} \leftarrow \mathsf{AddRoundKey}(\{[\![x_{r,c}]\!]\}, \{[\![k^{(10)}_{r,c}]\!]\})$
13: Servers locally perform finalization step to obtain $\{[\![z_i]\!]\}_{i=0,\dots,127}$
14: return $\{[\![z_i]\!]\}_{i=0,\dots,127}$
```

B PROOF OF BATCH VERIFICATION PROTOCOL

We now prove security of Protocol 2.

Theorem B.1 (Theorem 2.3, restated). *Protocol 2 (*Verify*) securely realizes the functionality $\mathcal{F}_{\mathsf{verify}}$ (see Fig. 3), in the ($\mathcal{F}_{\mathsf{weakMult}}$, $\mathcal{F}_{\mathsf{coin}}$, $\mathcal{F}_{\mathsf{rand}}$)-hybrid model. The failure probability in the simulation is at most $(m + 2\log N)/|\mathbb{F}|$.*

Proof. We begin by proving the base case when $N = 1$, namely, Protocol 1.

Proposition B.2. *Protocol 1 securely realizes $\mathcal{F}_{\mathsf{verify}}$ for a single multiplication triple.*

Proof. The simulator, $\mathcal{S}$, receives the corrupted parties' shares $[\![x]\!]^C, [\![y]\!]^C, [\![z]\!]^C$ from $\mathcal{F}_{\mathsf{verify}}$, and receives the outcome $b \in \{$**accept**, **abort**$\}$. It emulates $\mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}_{\mathsf{weakMult}}$, receiving shares $[\![x']\!]^C$, $[\![r]\!]^C$ and $[\![z']\!]^C$, plus an additive error d, and then sends a random $t \leftarrow \mathbb{F}$ for $\mathcal{F}_{\mathsf{coin}}$. It simulates the first Reconst by sending a random value ρ. For the second $\mathcal{F}_{\mathsf{weakMult}}$, it receives the shares $[\![\sigma]\!]^C$, together with another additive error f. For the second Reconst, if $b =$ **accept**, $d = 0$ and $f = 0$ then $\mathcal{S}$ opens σ by sending honest shares corresponding to the secret 0, and sends **continue** to $\mathcal{F}_{\mathsf{verify}}$. Otherwise, it samples honest shares corresponding to a random σ, sends these to the adversary and sends **abort** to $\mathcal{F}_{\mathsf{verify}}$.

First, notice that in the real world, ρ is statistically close to uniform, because of the tx' term that masks x. Secondly, if $e = z - xy$ is the error in the triple, then we have:

$$z + tz' - \rho y = z - xy + t(z' - x'y) = e + td$$

So, if the triple is correct and the adversary chooses $d = 0, f = 0$ then we always have $\sigma = 0$ in both real and ideal worlds, and furthermore the output in both cases will be **accept**. On the other hand, if there are any errors then in the real world we have $\sigma = (e + td)r + f$. If e is non-zero, then $e + td$ is non-zero except with probability $1/|\mathbb{F}|$, since d is fixed before the sampling of t. It follows that σ is statistically close to uniform, since r is uniformly random and unknown to the adversary. Finally, this implies that the protocol output will be **abort** except with probability $1/|\mathbb{F}|$, which is statistically close to the ideal world. □

Next, we analyze the VerifyDotProduct procedure with an inductive argument. Namely, we show that if the recursive call to VerifyDotProduct of length $N/2$ securely realizes the functionality, then so does the main procedure.

Proposition B.3. *Suppose that* VerifyDotProduct *on input of an inner product triple of length $N/2$ securely implements $\mathcal{F}_{\mathsf{verify}}$, with $m = 1$ and length $N/2$. Then,* VerifyDotProduct *securely implements $\mathcal{F}_{\mathsf{verify}}$ with $m = 1$ and length N. The failure probability in the simulation is $2/|\mathbb{F}|$.*

Proof. We construct a simulator, $\mathcal{S}$, as follows. $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{verify}}$ the corrupted shares of the triple $[\![\vec{x}]\!]^C, [\![\vec{y}]\!]^C, [\![z]\!]^C$, and the result $b \in \{$**accept**, **abort**$\}$. $\mathcal{S}$ computes the shares of f_i and g_i, and uses these to simulate $\mathcal{F}_{\mathsf{weakDotProduct}}$. It receives the adversary's shares $h(1), h(2)$, and locally computes the shares of $h(0)$, to define shares of the polynomial $[\![h(X)]\!]^C$. It also receives from $\mathcal{A}$ errors, which define via interpolation an error polynomial $e(X)$ such that $h(X) = \vec{f}(X) \cdot \vec{g}(X) + e(X)$.

Next, $\mathcal{S}$ sends a random $r \leftarrow \mathbb{F}$ to $\mathcal{A}$. It then emulates the recursive call to VerifyDotProduct; if $b =$ **abort** or $e(X) \neq 0$, it sends **abort** to the adversary, followed by **abort** to the length-N $\mathcal{F}_{\mathsf{verify}}$. Otherwise, it sends **accept** to the adversary; if it responds with continue, then send **accept** to $\mathcal{F}_{\mathsf{verify}}$, otherwise send **abort**.

We now argue indistinguishability. In the real world, if the inner VerifyDotProduct check succeeds then $h(r) = \vec{f}(r) \cdot \vec{g}(r)$. Since $h(X)$ is degree at most 2, this implies that $h(X)$ and $\vec{f}(X) \cdot \vec{g}(X)$ are equal as polynomials, except with probability $2/|\mathbb{F}|$. Since $h(0) + h(1) = z$, by construction, it follows that, except with negligible probability, if the protocol accepts then $z = \vec{f}(0) \cdot \vec{g}(0) + \vec{f}(1) \cdot \vec{g}(1) = \vec{x} \cdot \vec{y}$, as required.

Meanwhile, in the ideal world, the simulator always aborts if the triple is incorrect, or if $e(X) \neq 0$. The only possible differences between the two worlds are the cases: (i) the triple is correct, but $e(X) \neq 0$ and VerifyDotProduct accepts, or (ii) the triple is incorrect, but the real protocol accepts. Each of these cases would require VerifyDotProduct to accept in the real world, even though $h(X) \neq \vec{f}(X) \cdot \vec{g}(X)$. As argued above, this happens with probability at most $2/|\mathbb{F}|$. □

Proposition B.4. *If at least one triple input to Protocol 2 is incorrect, then so is the input to* VerifyDotProduct*, except with probability at most $m/|\mathbb{F}|$.*

Proof. Suppose that $z_i = \vec{x}_i \cdot \vec{y}_i + \delta_i$, and at least one $\delta_i \neq 0$. Then, if

$$(\vec{x}_0, r\vec{x}_1, \dots, r^{m-1}\vec{x}_{m-1}) \cdot (\vec{y}_0, \dots, \vec{y}_{m-1}) = \sum_{i=0}^{m-1} r^i z_i \,,$$

then it holds that

$$(1, r, \dots, r^{m-1}) \cdot (\delta_0, \dots, \delta_{m-1}) = 0 \,.$$

Viewing the δ_i's as coefficients of a non-zero, degree $m-1$ polynomial, this holds with probability at most $m/|\mathbb{F}|$, for a random r. □

The claim and final bound in the theorem follows by a hybrid argument over the $\log N$ recursive calls to VerifyDotProduct and the final base case. □

C RANDOM ONE-HOT VECTOR PROTOCOL FOR LENGTH 16

We propose a protocol to securely compute the ideal functionality $\mathcal{F}_{\mathsf{RandOHV}}$ with length-16 output in Protocol 9. Compared to Protocol 5, it has the same communication complexity but fewer rounds. The fundamental idea is based on the two-party Unitv-prep protocol for secure random unit vectorization protocol proposed in [7]. However, our proposed approach differs in that it allows the multi-party setting and it outputs sharings that are suitable for our construction.

The idea behind the construction is based on the fact that for a single random bit b, the pair $(b \oplus 1, b)$ forms a one-hot vector of length 2. Additionally, two one-hot vectors of length t can be tensor-multiplied to generate a one-hot vector of length $2t$.

In the proposed approach, the constructed one-hot vector $\boldsymbol{e}^{(r)}$ satisfies the following equation

$$\boldsymbol{e}_j^{(r)} = \bigwedge_{0 \le i \le 3} (j[i] \oplus r[i] \oplus 1), \tag{4}$$

where $j[i]$ (resp., $r[i]$) represents the i-th bit of $j \in \mathbb{Z}_{16}$ (resp., $r \in \mathbb{Z}_{16}$).

Note that, by the distributive property, the terms on the right-hand side for any $j \in \mathbb{Z}_{16}$ can be expressed as the sum of partial products of $\{r_0, r_1, r_2, r_3\}$. The proposed protocol achieves the one-hot encoding by generating random shares of $r[i] \in \mathbb{F}_2$ and securely computing all their partial products using Eq. (4).

Lemma C.1. *The protocol* RndOhv *in Protocol 9 securely computes* $\mathcal{F}_{\mathsf{RandOHV}}$ *for* $k = 4$ *with abort in the* $\{\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{weakMult}}, \mathcal{F}_{\mathsf{verify}}\}$*-hybrid model in the presence of a malicious adversary under the honest majority setting.*

Proof. **Simulation of** RndOhv**.** $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{rand}}$ and receives from $\mathcal{A}$ the share $[\![r_3]\!]^{\mathcal{C}}, \dots, [\![r_0]\!]^{\mathcal{C}}$ held by corrupted parties. For 11 invocations of Mult, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{weakMult}}$ and sends $\mathcal{A}$ corrupt parties' input shares $([\![r_i]\!]^{\mathcal{C}}, [\![r_j]\!]^{\mathcal{C}})$ for all $3 \ge i > j \ge 0$, $([\![r_i]\!]^{\mathcal{C}}, [\![r_j r_k]\!]^{\mathcal{C}})$ for all $3 \ge i > j > k \ge 0$, and $([\![r_3 r_2]\!]^{\mathcal{C}} [\![r_1 r_0]\!]^{\mathcal{C}})$. $\mathcal{S}$ receives from $\mathcal{A}$ the pairs of the error and shares $(d_{ij}, [\![r_i r_j]\!]^{\mathcal{C}})$ for all $3 \ge i > j \ge 0$, $(d_{ijk}, [\![r_i r_j r_k]\!]^{\mathcal{C}})$ for all $3 \ge i > j > k \ge 0$, and $(d, [\![r_3 r_2 r_1 r_0]\!]^{\mathcal{C}})$. $\mathcal{S}$ computes $[\![\boldsymbol{e}_j^{(r)}]\!]^{\mathcal{C}}$ for $j \in [0, 15]$ using the above shares held by corrupted parties. $\mathcal{S}$ sends $[\![r]\!]^{\mathcal{C}} := [\![r_{k-1}\| \dots \| r_0]\!]^{\mathcal{C}}$ and $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{C}} := [\![\boldsymbol{e}_0^{(r)} \| \dots \| \boldsymbol{e}_{15}^{(r)}]\!]^{\mathcal{C}}$ to $\mathcal{F}_{\mathsf{RandOHV}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verify}}$ and sends $\mathcal{A}$ the tuples consisting of the error and multiplication triple, $(d_{ij}, ([\![r_i]\!]^{\mathcal{C}}, [\![r_j]\!]^{\mathcal{C}} [\![r_i r_j]\!]^{\mathcal{C}}))$ for all $3 \ge i > j \ge 0$, $(d_{ijk}, ([\![r_i]\!]^{\mathcal{C}}, [\![r_j r_k]\!]^{\mathcal{C}} [\![r_i r_j r_k]\!]^{\mathcal{C}}))$ for all $3 \ge i > j > k \ge 0$, and $(d, ([\![r_3 r_2]\!]^{\mathcal{C}}, [\![r_1 r_0]\!]^{\mathcal{C}}, [\![r_3 r_2 r_1 r_0]\!]^{\mathcal{C}}))$. If there exists a non-zero error, $\mathcal{S}$ sets b = **abort**, and otherwise $\mathcal{S}$ sets b = **accept**. If b = **accept** and $\mathcal{A}$ replies **continue** , $\mathcal{S}$ proceeds to the next step. Otherwise, $\mathcal{S}$ sends **abort** to $\mathcal{F}_{\mathsf{RandOHV}}$ and aborts.

We now show that the ideal execution and the real execution are indistinguishable. The view of $\mathcal{A}$ consists of the corrupt parties' input shares to $\mathcal{F}_{\mathsf{weakMult}}$, which are computed the same since they are obtained through linear operations from $[\![r_3]\!]^{\mathcal{C}}, \dots, [\![r_0]\!]^{\mathcal{C}}$. We also show that the output shares of all parties are distributed the same in both executions. The corrupted parties output shares in the ideal world are computed the same way as those in the real world. For the honest parties' shares, they are determined by using $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{C}}$ and $\boldsymbol{e}^{(r)}$ in the ideal world conditioned on $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ is a shared one-hot vector of r. In the real world, $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ is computed from $[\![r_3]\!]^{\mathcal{H}}, [\![r_2]\!]^{\mathcal{H}}, [\![r_1]\!]^{\mathcal{H}}, [\![r_0]\!]^{\mathcal{H}}$ as defined in Eq. (4) which satisfies $\boldsymbol{e}^{(r)}$ is a one-hot vector of r. Here, $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ is distributed uniformly since it is computed via linear combination of corrupted parties' shares of output from $\mathcal{F}_{\mathsf{weakMult}}$. □

Protocol 9 Random One-hot Vector (RndOhv) with Length 16

```
Functionality: $(\{[\![r_i]\!]\}_i, \{[\![\boldsymbol{e}_j^{(r)}]\!]\}_j) \leftarrow \mathcal{F}_{\mathsf{RandOHV}}(\perp)$
Input: $\perp$
Output: Shared random boolean values $([\![r_3]\!], [\![r_2]\!], [\![r_1]\!], [\![r_0]\!])$ for
    $r_0, r_1, r_2, r_3 \in \{0,1\}$ and the corresponding shared one-hot vector
    $[\![\boldsymbol{e}^{(r)}]\!]$ for $r = \sum_{i=0}^{3} 2^i r_i$, where $|\boldsymbol{e}^{(r)}| = 16$
Subfunctionality: $\mathcal{F}_{\mathsf{rand}}$
1: $[\![r_k]\!] \leftarrow \mathcal{F}_{\mathsf{rand}}(\perp)$ for $k \in [0,3]$
2: $[\![r_i r_j]\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![r_i]\!], [\![r_j]\!])$ for all $3 \ge i > j \ge 0$
3: $[\![r_i r_j r_k]\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![r_i]\!], [\![r_j r_k]\!])$ for all $3 \ge i > j > k \ge 0$
4: $[\![r_3 r_2 r_1 r_0]\!] \leftarrow \mathcal{F}_{\mathsf{weakMult}}([\![r_3 r_2]\!], [\![r_1 r_0]\!])$
                                                    ▹2 offline rounds, 11 bits
5: Servers locally compute $[\![\boldsymbol{e}_j^{(r)}]\!]$ from the shares of the products as in
    Eq.(4), for $j \in [0,15]$
6: Execute $\mathcal{F}_{\mathsf{verify}}$ for the following multiplication triplets:
    (1) $([\![r_i]\!], [\![r_j]\!], [\![r_i r_j]\!])$ for all $3 \ge i > j \ge 0$
    (2) $([\![r_i]\!], [\![r_j r_k]\!], [\![r_i r_j r_k]\!])$ for all $3 \ge i > j > k \ge 0$
    (3) $([\![r_3 r_2]\!], [\![r_1 r_0]\!], [\![r_3 r_2 r_1 r_0]\!])$
7: return $\{[\![r_i]\!]\}_{0 \le i \le 3}, \{[\![\boldsymbol{e}_j^{(r)}]\!]\}_{0 \le j \le 15}$
```

Reducing the number of multiplication checks. Instead of verifying all 11 AND gates separately, we observe that it suffices to check 2 multiplications over a sufficiently large extension field $GF(2^k) = GF(2)[X]/f(X)$:

The first multiplication verifies all the pairwise products $r_i r_j$:

$$\begin{aligned}(r_0 + r_1 X + r_2 X^2) \cdot (r_3 + r_0 X^3 + r_1 X^6) &= \\ &r_0 r_3 + r_1 r_3 X + r_2 r_3 X^2 \\ &+ X^3 (r_0 + r_0 r_1 X + r_0 r_2 X^2) \\ &+ X^6 (r_0 r_1 + r_1 X + r_1 r_2 X^2) \,.\end{aligned}$$

The second multiplication verifies the remaining products:

$$(r_0r_1 + r_0r_3X + r_1r_3X^2) \cdot (r_2 + r_3X^3 + r_2r_3X^6) = r_0r_1r_2 + r_0r_2r_3X + r_1r_2r_3X^2 + X^3(r_0r_1r_3 + r_0r_3X + r_1r_3X^2) + X^6(r_0r_1r_2r_3 + r_0r_2r_3X + r_1r_2r_3X^2)\,.$$

D DEFERRED PROOFS

D.1 Proof of Lemma 3.1

Lemma D.1 (Lemma 3.1, restated). *The protocol* Inv *in Protocol 3 securely computes $\mathcal{F}_{\mathsf{Inv}}$ with abort in the $\left\{\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle\to[\![\cdot]\!]}, \mathcal{F}_{\mathsf{weakMult}}, \mathcal{F}_{\mathsf{verify}}\right\}$-hybrid model in the presence of a malicious adversary under the honest majority setting.*

Proof. $\mathcal{S}$ receives $[\![x]\!]^C$ from $\mathcal{F}_{\mathsf{Inv}}$ and computes $[\![v_x]\!]^C$ through Step 1–7 using $[\![x]\!]^C$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle\to[\![\cdot]\!]}$ and receives $\langle\langle v\rangle\rangle^C$, $[\![v^{-1}]\!]^C$ and $[\![v]\!]^C$ from $\mathcal{A}$, and defines the error $d_v = [\![v_x]\!]^C \oplus [\![v]\!]^C$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{weakMult}}$, computes $[\![a_h]\!]^C$ and $[\![a_h]\!]^C \oplus [\![a_\ell]\!]^C$ using $[\![x]\!]^C$, and sends $([\![a_h]\!]^C, [\![v^{-1}]\!]^C), ([\![a_h]\!]^C \oplus [\![a_\ell]\!]^C, [\![v^{-1}]\!]^C)$ to $\mathcal{A}$. $\mathcal{S}$ receives $(d_1, [\![a'_h]\!]^C), (d_2, [\![a'_\ell]\!]^C)$ from $\mathcal{A}$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{verify}}$ and sends $\mathcal{A}$ the tuples consisting of the error and multiplication triple;

- $(d_v, ([\![a_h]\!], [\![a_\ell]\!], [\![a_h \times a_\ell]\!]))$
- $(d_1, ([\![a_h]\!], [\![v^{-1}]\!], [\![a'_h]\!]))$
- $(d_2, ([\![a_h \oplus a_\ell]\!], [\![v^{-1}]\!], [\![a'_\ell]\!]))$

Note that the first triple $([\![a_h]\!], [\![a_\ell]\!], [\![a_h \times a_\ell]\!])$ can indirectly prove that $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle\to[\![\cdot]\!]}$ outputs the correct $[\![v]\!]$, that is, c is correctly computed. Here, $[\![v^{-1}]\!]$ is computed locally and we don't need to verify.

If there exists a non-zero error, $\mathcal{S}$ sets $b =$ **abort**, and otherwise $\mathcal{S}$ sets $b =$ **accept**. If $b =$ **accept** and if $\mathcal{A}$ replies **continue**, $\mathcal{S}$ proceeds to the next step. Otherwise, $\mathcal{S}$ sends **abort** to $\mathcal{F}_{\mathsf{Inv}}$ and aborts.

$\mathcal{S}$ computes $[\![x^{-1}]\!]^C$ using $[\![a'_h]\!]^C$ and $[\![a'_\ell]\!]^C$. $\mathcal{S}$ sends $[\![x^{-1}]\!]^C$ to $\mathcal{F}_{\mathsf{Inv}}$.

We state why the ideal execution is indistinguishable from the real execution. The view of the adversary consists of the corrupt parties' shares of the first inputs to multiplicative inversions $[\![a_h]\!]$ and $[\![a_h \oplus a_\ell]\!]$, but these are uniformly random in both executions because they are obtained by applying a non-zero affine map to $[\![x]\!]^C$. We also need to show that the output shares of all parties are distributed the same in both executions. The corrupted parties' output shares are the same in both executions. For the honest parties' output shares in the ideal execution, they are sampled at random conditioned on that they can be reconstructed to x^{-1}. In the real execution, the honest parties' output share $[\![y]\!]^{\mathcal{H}}$ is obtained as $[\![x^{-1}]\!]^{\mathcal{H}}$ as the correctness was shown in Sect. 2.1.1, and it is uniformly distributed since it is computed by applying a non-zero affine map to the output of multiplications that were sampled uniformly. □

D.2 Proof of Lemma 3.2

Lemma D.2 (Lemma 3.2, restated). *The protocol* LUT *in Protocol 4 securely computes $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]\to\langle\langle\cdot\rangle\rangle}$ in the $\{\mathcal{F}_{\mathsf{RandOHV}}, \mathcal{F}_{\mathsf{zero}}\}$-hybrid model in the presence of a malicious adversary.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator, $\mathcal{S}$, to simulate the honest parties' behaviour in the real execution.

Simulation of LUT. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{RandOHV}}$ and receives $[\![r]\!]^C$ and $[\![\boldsymbol{e}_j^{(r)}]\!]^C$ from $\mathcal{A}$, and defines $\langle\langle r\rangle\rangle^C = \mathsf{ToAdditive}([\![r]\!]^C)$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{zero}}$ and receives $[\![0]\!]^C$ from $\mathcal{A}$. $\mathcal{S}$ receives $\langle\langle c\rangle\rangle^C$ from the adversary and computes $\langle\langle v\rangle\rangle^C = \langle\langle c\rangle\rangle^C \oplus \langle\langle r\rangle\rangle^C \oplus \langle\langle 0\rangle\rangle^C$. $\mathcal{S}$ samples at random a set of shares $\langle\langle c\rangle\rangle^{\mathcal{H}}$ held by honest parties and sends them to the adversary. $\mathcal{S}$ computes the set of shares $[\![t]\!]^C := \bigoplus_{j=0}^{n-1}[\![\boldsymbol{e}_j^{(r)}]\!]^C \cdot T_{c\oplus j}$ and $[\![v]\!]^C := [\![r]\!]^C \oplus c$. $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle\to[\![\cdot]\!]}$ the corrupted parties' input shares $\langle\langle v\rangle\rangle^C$, and the outputs shares $[\![t]\!]^C, [\![v]\!]^C$.

We now argue why the ideal execution is indistinguishable from the real execution. The view of the adversary consists of the honest parties' shares of the reconstructed c, but these are uniformly random in both executions, thanks to the masking with $\langle\langle 0\rangle\rangle$. We also need to show that the output shares of all parties are distributed the same in both worlds. The corrupted parties' shares are computed exactly the same way in both executions. For the honest parties' shares, in the ideal world they are sampled at random conditioned on $t = T_v$. In the real protocol, since $\boldsymbol{e}_j^{(r)}$ has a 1 in position r, we have $t = T_{v\oplus r\oplus r} = T_v$. Furthermore, since the shares $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}, [\![r]\!]^{\mathcal{H}}$ sampled by $\mathcal{F}_{\mathsf{RandOHV}}$ are sampled uniformly, the output shares $[\![t]\!]^{\mathcal{H}}, [\![v]\!]^{\mathcal{H}}$ are also uniformly distributed, since they are each obtained by applying a non-zero affine map to $[\![\boldsymbol{e}^{(r)}]\!]^{\mathcal{H}}$ and $[\![r]\!]^{\mathcal{H}}$. □

D.3 Proof of Lemma 3.4

Lemma D.3 (Lemma 3.4, restated). *Protocol 7 securely realizes the functionality $\mathcal{F}_{\mathsf{AES}}$ in the $(\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}, \mathcal{F}_{\mathsf{verify}})$-hybrid model with malicious security.*

Proof. We construct a simulator, $\mathcal{S}$, as follows. First, $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{AES}}$ the corrupted parties' shares $[\![x]\!]^C$ and $[\![k^{(i)}]\!]^C$, and defines $\langle\langle k^{(i)}\rangle\rangle^C = \mathsf{ToAdditive}([\![k^{(i)}]\!]^C)$. For the first set of calls to $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]\to\langle\langle\cdot\rangle\rangle}$, $\mathcal{S}$ sends to $\mathcal{A}$ the appropriate shares of x_b^0, and receives the output shares $\langle\langle y_b^0\rangle\rangle^C$. For each subsequent round, for $i = 1, \ldots, 9$, $\mathcal{S}$ does as follows:

- Apply the linear layer L_i to the corrupted parties' shares to obtain $\langle\langle x^i\rangle\rangle^C$
- Receive $\langle\langle \hat{x}^i\rangle\rangle^C$ from $\mathcal{A}$, as input to $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$
- Define the error $\delta^i = \bigoplus_{j\in C}(\langle\langle x^i\rangle\rangle^j \oplus \langle\langle \hat{x}^i\rangle\rangle^j)$
- Receive the adversary's output sharings $\langle\langle y^i\rangle\rangle^C, [\![x_b^i]\!]^C$ for $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$
- Compute the shares $[\![y^{i-1}]\!]^C$ according to the protocol

Finally, $\mathcal{S}$ emulates Reshare, and computes the corresponding error δ^{10} in the new sharing of y^9. To emulate $\mathcal{F}_{\mathsf{verify}}$, $\mathcal{S}$ first sends to $\mathcal{A}$ the corresponding shares of the triples. Then, if any δ^i is non-zero, $\mathcal{S}$ sends **abort** to $\mathcal{A}$ and aborts; otherwise, $\mathcal{S}$ sends **accept**, and if $\mathcal{A}$ responds with **continue**, $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{AES}}$ the corrupted parties' shares of the outputs, z.

We claim that the ideal execution is distributed identically to that of the real execution. Note that the real protocol aborts if any of the S-box input/output pairs are incorrect; otherwise, the output z must be the result of a correct AES evaluation. In the ideal execution, the protocol aborts if any error δ^i is non-zero. Since δ^i is the sum of all corrupt parties' shares of the x^i value which was meant to be input into $\mathcal{F}_{\mathsf{LUT}}$, and $\hat{x}^i$ was was used as input, any non-zero δ^i means that an incorrect S-box input was used in round i. Since this input is used to derive the sharings $[\![x^i]\!]$, this will cause the S-box verification for round $i-1$ to fail, and the protocol will abort. □

E IMPROVED PROTOCOL FOR $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$

We present the improved protocol for $\mathcal{F}_{\mathsf{LUT}}^{\langle\langle\cdot\rangle\rangle}$ in Protocol 10. Since the protocol operates entirely on $[\![\cdot]\!]$-shared data, its security is straightforward and we omit the proof of the following.

Lemma E.1. *Protocol 10 securely realizes the functionality $\mathcal{F}_{\mathsf{LUT}}^{[\![\cdot]\!]}$ with malicious security.*

Protocol 10 Table Lookup of size $N = 2^k$ in replicated sharing

Functionality: $[\![T_v]\!] \leftarrow \mathcal{F}_{\mathsf{LUT}}([\![v]\!], T)$
Input: Share $[\![v]\!]$ of $v \in GF(2^k)$, table $T : GF(2^k) \to GF(2^\ell)$
Output: Share $[\![T_v]\!]$ of the value of T at v
Subfunctionality: $\mathcal{F}_{\mathsf{RandOHV}}$

```
1: Call $\mathcal{F}_{\mathsf{RandOHV}}(k/2)$ twice to get $(\{[\![r_i]\!]\}_{i=0}^{k/2-1}, \{[\![\boldsymbol{e}_j^{(r)}]\!]\}_{j=0}^{\sqrt{N}-1})$
   and $(\{[\![r'_i]\!]\}_{i=0}^{k/2-1}, \{[\![\boldsymbol{e}_j^{(r')}]\!]\}_{j=0}^{\sqrt{N}-1})$
2: $[\![r]\!] := ([\![r_0]\!], \dots, [\![r_{k/2-1}]\!], [\![r'_0]\!], \dots, [\![r'_{k/2-1}]\!])$
3: $c \leftarrow \mathsf{Reconst}([\![v]\!] + [\![r]\!])$        ▷1 round, $k$ bits
   ▷$\vec{f}^{(0)}, \vec{f}^{(1)} \in \{0,1\}^N$
4: $[\![\vec{f}^{(0)}]\!] := ([\![\boldsymbol{e}_0^{(r)}]\!], \dots, [\![\boldsymbol{e}_0^{(r)}]\!], \dots, [\![\boldsymbol{e}_{\sqrt{N}-1}^{(r)}]\!], \dots, [\![\boldsymbol{e}_{\sqrt{N}-1}^{(r)})]\!]$
5: $[\![\vec{f}^{(1)}]\!] := ([\![\boldsymbol{e}_0^{(r')}]\!], \dots, [\![\boldsymbol{e}_{\sqrt{N}}^{(r')}]\!], \dots, [\![\boldsymbol{e}_0^{(r')}]\!], \dots, [\![\boldsymbol{e}_{\sqrt{N}-1}^{(r')})]\!]$
6: $[\![\vec{g}^{(0)}]\!] := (T_c \cdot [\![\vec{f}_0^{(0)}]\!], \dots, T_{c \oplus (N-1)} \cdot [\![\vec{f}_{N-1}^{(0)}]\!])$
7: $[\![v]\!] \leftarrow \mathcal{F}_{\mathsf{weakDotProduct}}([\![\vec{g}^{(0)}]\!], [\![\vec{f}^{(1)}]\!])$        ▷1 round, $k$ bits
8: Run $\mathcal{F}_{\mathsf{verify}}$ on input $([\![\vec{g}^{(0)}]\!], [\![\vec{f}^{(1)}]\!], [\![v]\!])$
9: return $[\![v]\!]$
```