Additions, Multiplications, and the Interaction In-Between: Optimizing MPC Protocols via Leveled Linear Secret Sharing (Full Version)*

Andreas Brüggemann, Thomas Schneider, and Maximilian Stillger
Technical University of Darmstadt, Germany
{brueggemann,schneider,stillger}@encrypto.cs.tu-darmstadt.de

***Abstract*—Secure multiparty computation (MPC) enables distrusting parties in a distributed system to compute on their private inputs without compromising their privacy. For many secret-sharing-based approaches, including some of today's most efficient MPC protocols, there is a pattern where two shared values are locally multiplied into some intermediate representation that is immediately and interactively translated back into sharings of the product. The intermediate representation is often still a full-fledged but different secret sharing scheme. This has been used to efficiently compute dot products by computing the sum of all intermediate products and then interactively translating only the sum instead of translating each individual product. Beyond that, the intermediate representation or secret sharing scheme has mostly been seen only as a necessary interim step, leaving most of its potential untapped.**

We change that by proposing the paradigm of *leveled linear secret sharing*, which allows dynamic switching between the original secret sharing and the previously only intermediate one more freely, while enabling arbitrary linear computations in any of the domains. Prior multiplications are split into a non-interactive multiplication that switches from one to the other secret sharing, and an interactive upgrade back to the original secret sharing domain. The upgrade now does not necessarily follow each multiplication immediately, but just needs to be placed *somewhere* before the next multiplication is computed, possibly upgrading the linear aggregation of many multiplications' results. We apply this idea to improve three-party computation on replicated sharings (CCS'16), n-party BGW-style protocols (STOC'88), and masked secret sharing protocols such as ABY2.0 (USENIX Security'21). We build a novel optimizer that optimally selects which gate of a circuit is evaluated in which domain. With that, we improve communication by 10–37% for many circuits. Furthermore, we implement our generalization for replicated sharing, measure run time improvements of mostly 10–26% in a LAN, and make a full implementation of the protocol and our novel optimizer publicly available.

I. Introduction

Secure Multiparty Computation (MPC) is a valuable tool for privacy-preserving computation. It enables mutually distrusting parties to jointly compute functions on their collective inputs while keeping their individual inputs private, e.g., if they contain personally identifiable information or trade secrets. Possible applications are privacy-preserving machine learning (PPML) [1], [2], [3], [4], [5], [6], private analysis of large (social) graphs [7], [8], [9], [10], private database operations [11], [12], and many more. To achieve high throughput, a popular choice is to rely on MPC protocols based on linear secret sharing, such as; [13], [14], [15], [16] based on Shamir's secret sharing [17]; [18], [19], [20] based on additive secret sharing, [21], [22], [23], [24], [25] based on replicated secret sharing; or [26], [27], [28], [29] based on masked secret sharing. Optimization of such generic protocols that securely evaluate any arithmetic/binary circuit is important, as any optimization translates to numerous possible applications and more specialized protocols built on top of the generic one.

Real-world functionalities often consist of building blocks with different requirements. For instance, a linear layer in a convolutional neural network directly translates to an arithmetic circuit which can be evaluated using arithmetic secret sharing, while nonlinear layers are preferably represented as a binary circuit, evaluated in binary secret sharing. Depending on the circuit topology in the binary domain, using garbled circuits [30] provides yet another option with different tradeoffs. Hence, *hybrid MPC* [31], [28], [1], [32], [33], [34] combines MPC in different domains into one protocol, using specific conversion protocols to switch between domains, aiming to use for each part of the computation the specific, optimal primitives. The decision on which part of a functionality is best computed in which domain induces an optimization problem that has received significant attention, yielding many automated optimizers [35], [36], [37], [38], [39].

Yet, even within the same domain (arithmetic or binary) and when only using linear secret sharing, it can be beneficial to consider multiple types of secret sharings. This has been done implicitly as early as in the BGW-protocol [13] based on Shamir's secret sharing [17]. Here, a secret value x is represented by shares $f(1), f(2), f(3), \ldots, f(n)$ for a random polynomial f with $f(0) = x$ and a degree $d \approx n/2$ such that half of all shares are required to recover x. For multiplying secret shared values x, y represented by polynomials f, g, each party i simply computes $h(i) := f(i) \cdot g(i)$, inducing a polynomial h of degree $2d$, representing the product $h(0) = f(0) \cdot g(0) = x \cdot y$. The expensive, interactive part of the protocol then is to *re-randomize* the polynomial and *reduce its degree* back to d. This is necessary as h is not truly random (subject to $h(0) = x \cdot y$) as it is the product of two polynomials, whereas randomness is crucial for the security of the protocol. Furthermore, any future multiplication would again increase the degree, now to a value where all parties'

*Please cite the conference version of this paper at NDSS 2027 [42].

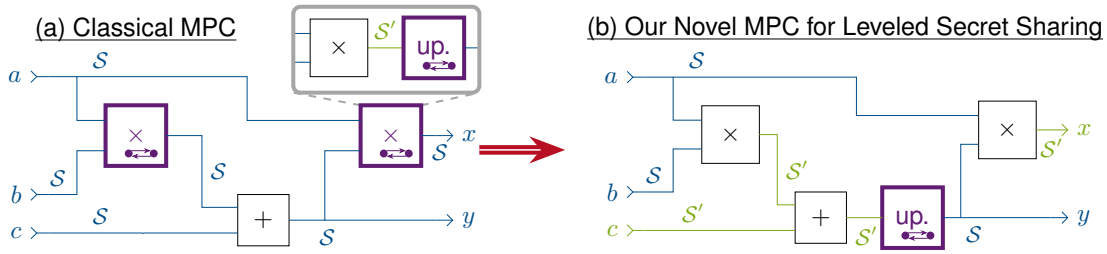


Figure 1: Classical MPC protocols based on a linear secret sharing scheme $\mathcal{S}$, as depicted on the left, require interaction for their multiplications. Routinely, each multiplication consists of a local multiplication, yielding some *intermediate* sharing $\mathcal{S}'$ which then is immediately and interactively upgraded back to $\mathcal{S}$. Instead, as shown on the right, we detach upgrading from local multiplication, allowing for linear computation, inputs, and outputs both in $\mathcal{S}/\mathcal{S}'$. "⇄" marks gates that require interaction.

shares are insufficient to reconstruct the secret. Still, as long as we do not multiply and, hence, there is no communication, we could hypothetically continue to work with higher degree polynomials, delaying re-randomization and degree reduction. This has been utilized by [40], 22 years after publication of BGW, to compute dot products of dimension ℓ more efficiently. Instead of naïve computation by ℓ BGW multiplications followed by addition, they *locally* compute shares of degree $2d$, first sum these up, and only then *interactively* re-randomize and reduce the degree for the single result instead of for all ℓ multiplications. This general nature of *locally* multiplying into another domain (such as double degree polynomials) and **then** *interactively* changing back to the original domain is not exclusive to BGW. The popular and highly efficient three-party protocol from [21] multiplies *replicated secret shares* locally, yielding an *additive secret sharing*, which it then immediately reshares to a replicated secret sharing. Again, this was only later exploited for dot products [41], adding multiple additive sharings before resharing only one value. In another line of work [27], [28], [29], *masked sharings* are locally multiplied to some intermediate sharings and then reshared, which is immediately also utilized for more efficient dot products. In Fig. 1a, the *classical* computation using this family of protocols is shown, using intermediate sharings only internally in multiplications (or dot products).

Still, the intermediate domain is not fully utilized in prior works. Dot products were investigated as an "obvious target" and also due to their high relevance for machine learning. To fully unleash the power of the intermediate domain, it should be viewed as an alternative domain that just does not allow for further multiplication before changing back to the *main* domain, but still allows for other computation. This not only enables the addition of the direct results of multiplications, but also multiplications by constants in the alternative domain, using individual products in multiple sums, and even providing inputs in either of the domains. An example is provided in Fig. 1b, demonstrating how a mixed use of both secret sharing domains enables decreasing the number of required interactive steps while showcasing inputs and outputs in both domains. Yet, this optimization requires proper formalization, security analysis, and a tool to automatically determine which part of a circuit is best evaluated in which of the domains, similar to prior optimizations in hybrid MPC. This leads over to the contributions of our work.

A. Our Contributions

MPC Based on Leveled Linear Secret Sharing (LLSS). We formalize the notion of *leveled linear secret sharing* (LLSS), consisting of two linear secret sharing schemes $\mathcal{S}, \mathcal{S}'$, where sharings in $\mathcal{S}$ can be multiplied non-interactively, yielding a sharing in $\mathcal{S}'$. Based on that, we propose MPC based on such leveled scheme where linear operations, inputs, and outputs can be in either domain ($\mathcal{S}$ or $\mathcal{S}'$), multiplication switches from $\mathcal{S}$ to $\mathcal{S}'$, and an interactive upgrade operation converts from $\mathcal{S}'$ back to $\mathcal{S}$ (see Fig. 1b for an example). We give several concrete and provably secure instantiations based on the following secret sharing schemes and protocols, showcasing that our generic approach can be used as a starting point to optimize many existing MPC protocols:

- The BGW protocol [13]: Beyond computation on Shamir's secret sharing scheme [17] with threshold $t \approx n/2$, we consider the full-threshold variant of Shamir as an alternative domain $\mathcal{S}'$.
- MPC on replicated secret sharing [21]: We consider computation on replicated *and* full-threshold additive secret sharing.
- MPC based on *masked secret sharing* (e.g., as in ABY2.0 [28], ASTRA [27], Turbospeedz [26], and SWIFT [29]): We use the original masked secret sharing *and* the underlying secret sharing scheme for the masks (e.g., additive, replicated).

Optimal Sharing Assignment for LLSS. LLSS does not optimize MPC on a per-multiplication basis, but rather across the complete circuit to be computed. Hence, it is necessary to decide which parts of the circuit to evaluate in which secret sharing scheme to optimize communication. For this sharing assignment, we propose a novel optimization algorithm that minimizes the communication for securely evaluating a circuit. Our algorithm is based on finding a minimum cut, yielding not only polynomial, but also concretely efficient run time. The search space we consider is a superset of what could be hand-optimized by existing manually designed dot products. Hence, the optimizer can find more efficient

sharing assignments while also removing the need for manual optimization, even for dot products. Furthermore, it works in arithmetic and binary domains, whereas dot products are usually considered in the arithmetic domain only, and it also enables to have inputs and outputs in either secret-sharing domain for further optimization.

Extensive Evaluation, Including an Implementation for Leveled Replicated Secret Sharing [21]. Using our optimizer, we investigate communication improvements for protocols with replicated sharing (§III-C), Shamir sharing [17] (§III-D), and masked two-party secret sharing (§III-E) on a range of popular benchmarking circuits. Across all protocols, we improve (online[1]) communication by 2–37% depending on the circuit topology, with over 10% improvement for the majority of circuits. For large dot products, we even improve by $\approx 70\%$ compared to naïve dot product computation—an improvement which was also possible with existing dot product gates, but now, for the first time, is automated, not requiring to recognize and use dot products during circuit design manually. Our improvements are not the result of designing completely new protocols, but of our generic LLSS-approach that unleashes the full potential of multiple existing protocols. We also provide an implementation for [21] extended to our LLSS-variant and, based on that, observe a run time improvement of up to 25% on Boolean or even 90% on arithmetic circuits compared to the plain protocol of [21]. We make our implementations available at **https://encrypto.de/code/LLSS-MPC**, containing our optimizer as well as the LLSS extension of [21].

Obtaining Masked LLSS-MPC From Any MPC Protocol Based on Linear Secret Sharing. Multiple protocols, e.g., [28], [27], [26], utilize variants of *masked secret sharing*. We formalize the notion of masked secret sharing, yielding a compiler that builds a masked secret sharing MPC protocol with high online performance on top of any MPC protocol using linear secret sharing. This not only generalizes prior works, but also gives a straightforward generalization to LLSS, providing improvement to all existing works with masked secret sharing and a compiler to create more such protocols.

World Between Beaver Multiplication [43] and Masked Secret Sharing. Classical Beaver multiplication [43] can be phrased as masked LLSS-MPC that places upgrades immediately *before* multiplication inputs. In contrast, works on masked secret sharing [28], [27], [26], [29] correspond to our masked LLSS-MPC in the case of always immediately placing upgrades *after* multiplication outputs. As our LLSS-MPC enables us to freely place upgrades *anywhere* on the path between two multiplications, it is a generalization encompassing both approaches and the entire new world in-between, unifying and extending two popular prior paradigms (cf. §III-E3).

[1]For Shamir sharings [17] (§III-D) and masked sharings (§III-E), we also use preprocessing. For §III-D, our approach yields similar preprocessing improvements. For §III-E, this depends on the setting, and we provide possible preprocessing optimizations in §III-E2.

B. Related Work

Intermediate Secret-Sharing Domain in MPC Multiplications: Many MPC protocols based on linear secret sharing have multiplication sub-protocols that consist of a non-interactive multiplication step that yields the product in some intermediate linear secret-sharing domain, and a resharing/upgrade step switching back to the original sharing, e.g., [13], [21], [28], [15], [29]. This has been utilized to efficiently evaluate dot products at the price of a single multiplication by first adding many multiplication results and then just upgrading the sum [40], [41], [28], [29]. Our notion of MPC on leveled linear secret sharing captures this optimization, but is strictly more general as we enable *arbitrary* linear operations on multiplication outputs, yielding one or *more* intermediate results to be upgraded again. Furthermore, we open up the utilization of the intermediate domain even more by also enabling individual inputs and outputs to be from either the original or the intermediate domain. MAESTRO [44] also considers separate non-interactive multiplication and upgrade operations, performing linear computation more general than dot products before upgrading. Yet, MAESTRO uses this only as a building block for hand-optimizing AES evaluation within MPC using replicated secret sharing. Our approach works for multiple underlying secret sharings and generalizes to any application with an optimizer that always finds the best strategy for upgrading instead of requiring to optimize circuits by hand. ALKAID [45] considers the computation of XOR gates on outputs of AND gates before upgrading to hand-optimize parallel prefix adders in their scheme. Our approach is more general and does not require their hand-optimization, as we determine the optimal upgrading strategy automatically. MAESTRO and ALKAID also design and use novel optimized building blocks such as lookup table protocols for even higher efficiency, whereas our goal is to generically and automatically optimize existing circuits. We give a detailed comparison between our generic approach and the hand-optimized solutions for specific circuits by MAESTRO and ALKAID in §D-D. Finally, our masked LLSS construction also encompasses MPC from Beaver triples [43] as a special case (cf. §III-E3).

Leveled Homomorphic Encryption: The BGN [46] cryptosystem allows homomorphic evaluation of depth $L = 1$ circuits, where additions can be computed before and after a multiplication. More recently, leveled homomorphic encryption [47], [48] enables to evaluate a circuit of up to some maximum number of multiplication layers L. If more layers are required, expensive bootstrapping is required to enable further multiplications, making the scheme fully homomorphic. Our LLSS could somewhat be seen as an MPC equivalent of that for only $L = 1$ levels of multiplications before an interactive upgrade is required to enable further multiplications. In §III-F, we additionally discuss how our approach could be generalized to multiple layers. However, this is not the focus of our work and is limited to very few layers and, potentially, many parties with a very low corruption threshold.

Sharing Assignment Optimization: Multiple past works on hybrid MPC automatically combine different secret sharing types, e.g., Boolean, arithmetic, homomorphically encrypted, and Yao's garbled circuit [30] sharing, to optimize the performance of MPC computations that span many different kinds of operation types internally [49], [31], [1], [2], [28], [32], [33], [6]. To automatically assign sharing semantics and conversions between them to different components of a circuit, multiple optimization tools have been proposed. Multiple works [35], [38], [39] utilize integer linear programming (ILP) for searching the sharing assignment with minimal estimated protocol execution cost. Yet, solving an ILP is NP-hard, requiring high optimization cost and heuristic techniques such as segmenting a circuit into smaller parts and only solve these parts optimally. In [36], the circuit is segmented into smaller parts, which are optimized by enumerating all sharing assignments, again yielding high cost and suboptimal results. The ILP-based optimization in [37] can be done in polynomial time if only two choices of sharing semantics exist per an ILP relaxation. Yet, they do not specify an asymptotic complexity or concrete optimizer run times. Our sharing assignment could potentially be optimized by [37], but the concrete structure of our optimization problem allows for a simpler approach that relies neither on expensive ILP solving nor on complex relaxations and reductions to reach polynomial run time. In [50], heuristic optimization is used for good efficiency.

In contrast, our approach (§IV) is an algorithm customized to our specific share assignment problem, minimizing communication cost in MPC based on our novel notion of leveled linear secret sharing. The specific nature of our novel optimization problem enables to reduce it to a minimum cut problem, inheriting a run time complexity of at most $\mathcal{O}(|\mathcal{C}|^3)$ for circuit size $|\mathcal{C}|$, and we observe good concrete efficiency in §V.

C. Outline

After the preliminaries (§II), we introduce the notion of leveled linear secret sharing (LLSS) in (§III) and generalize multiple existing sharing schemes and MPC protocols to that while also formalizing MPC on masked secret sharing. Then, we show how to automatically utilize LLSS optimally (§IV), investigate improvements on practical circuits (§V), and conclude our paper (§VI). In the appendix, we give supplementary protocol details (§A), full security proofs (§B), details on our optimizer (§C), additional evaluation results (§D), and further pointers for future directions (§E).

II. Preliminaries

A. Notation

We consider n parties, denoted by $\mathcal{P}_1, \ldots, \mathcal{P}_n$. Given party $\mathcal{P}_i$, we sometimes refer to the next party $\mathcal{P}_{i+1}$ or the prior party $\mathcal{P}_{i-1}$ in cyclic fashion, i.e., $\mathcal{P}_{n+1} := \mathcal{P}_1$ and $\mathcal{P}_{1-1} := \mathcal{P}_n$ for ease of notation. For the set of integers $\{1, \ldots, k\}$ where $k \in \mathbb{N}$, we write $\mathbb{N}_{\leq k}$. As computation domain, we use a finite, commutative ring $\mathcal{R}$ which can be, e.g., $\mathbb{Z}_{2^k}$ or $\mathbb{F}_{p^k}$ for $k \in \mathbb{N}$ and prime p. This covers the arithmetic domain and, as a special case, also the binary domain $\mathcal{R} = \mathbb{F}_2$ where multiplication is a logic AND and addition is a logic XOR. We write $a \leftarrow\!\$\ A$ for sampling a uniformly random from set A.

B. Threshold Linear Secret Sharing Schemes

A (t, n)-threshold linear secret sharing scheme (LinSSS) $\mathcal{S}$ is defined via a share space $\mathcal{S}(\mathcal{R})$ over $\mathcal{R}$ and the procedures:

- The probabilistic procedure $\mathsf{share}(\cdot)$ has a secret input $v \in \mathcal{R}$ and returns shares $([v]^i_{\mathcal{S}})_{i \in \mathbb{N}_{\leq n}} \in \mathcal{S}(\mathcal{R})^n$.
- The deterministic procedure $\mathsf{rec}(\cdot)$ takes as input at least t shares $[v]^i_{\mathcal{S}}$ and reconstructs the secret value $v \in \mathcal{R}$.

We write $[v]_{\mathcal{S}} = ([v]^i_{\mathcal{S}})_{i \in \mathbb{N}_{\leq n}}$ for any *correct* sharing of $v \in \mathcal{R}$, i.e., a sharing where $\mathsf{rec}(\cdot)$ reconstructs v on any subset of at least t shares. Furthermore, we require correctness of $\mathsf{share}(\cdot)$, i.e., that $\mathsf{share}(v), v \in \mathcal{R}$ always outputs a correct sharing $[v]_{\mathcal{S}}$, and t-privacy, i.e., that no information about v can be learned from less than t shares from the output of $\mathsf{share}(\cdot)$. Finally, the sharing scheme must be linear: there must be a linear structure on the share space $\mathcal{S}(\mathcal{R})$ such that parties can perform addition of shared values, as well as addition and multiplication by constant non-interactively by some linear transformation of the shares. Throughout this paper, we sometimes omit explicitly defining $\mathsf{share}, \mathsf{rec}$ where they directly follow from the description of the sharing scheme.

The sharing procedure of a LinSSS can also be performed (partly) non-interactively via *pseudorandom secret sharing* (PRSS) as introduced in [51], [52], utilizing pseudorandom functions (PRFs). We will make use of this technique in the computational setting to save on network resources similar to [21], [16], [27], assuming that pairs of parties establish joint, random PRF keys in advance. These keys enable parties to non-interactively generate joint random values.

An easy example for a (n, n)-LinSSS is the GMW-style [18] additive secret sharing, which we later use for protocols:

Definition 1 ((n, n)-Additive Secret Sharing [18])**.** *(n, n)-additive secret sharing (nASS) is a (n, n)-threshold LinSSS, splitting a secret $v \in \mathcal{R}$ into n values $[v]^i_{nASS} \leftarrow\!\$\ \mathcal{R}$ subject to*

$$v = \sum_{i=1}^{n} [v]^i_{nASS}.$$

Throughout this paper, we will introduce further LinSSSs specific to some of the protocols where needed.

C. Threat Model

Throughout this work, we consider security against semi-honest/passive, non-adaptive adversaries, using the real-world/ideal-world simulation paradigm [53], [54]. In short, this assumes that the adversary decides before the protocol execution to corrupt a subset of parties (1 out of 3 parties for §III-C, up to $(n-1)/2$ out of n parties for §III-D, 1 out of 2 parties for §III-E with ABY2.0 [28]), learning these parties' inputs, internal states, and communication without altering their behavior. Simulation-based security directly implies correctness and privacy, i.e., that the adversary learns no information that it could not already infer from the corrupted parties' inputs

and outputs. Regarding communication, we assume that each pair of parties is connected with a secure channel.

D. MPC on Linear Secret Sharings

Throughout this work, we consider protocols secure against a semi-honest/passive, non-adaptive adversary, using the real-world/ideal-world simulation paradigm [53], [54]. The adversary can corrupt parties up to some corruption threshold, depending on each concrete protocol's setting. Formal details are provided alongside the security proofs in §B. We consider secure computation of arithmetic circuits over domain $\mathcal{R}$, which include Boolean circuits for the case of $\mathcal{R} = \mathbb{F}_2$. Values on all wires are encoded using a LinSSS (cf. §II-B).

It is common practice to divide the execution of MPC protocols into two phases. In the *preprocessing* (or offline) phase, the parties have not chosen their inputs yet but generate correlated randomness. This shifts large parts of the computational effort into the idle time of the parties' machines. Once all parties have their inputs, they run the *online* phase, which minimizes the latency of the computation from providing inputs to receiving results. In our work, we consider protocols without preprocessing (§III-C), with preprocessing (§III-D), or even *function-dependent* preprocessing which depends on the topology of the specific circuit evaluated later (§III-E). We emphasize that, in contrast to some function-dependent preprocessing works such as [28], [55], [29], our optimizations do not improve the online phase at the expense of more preprocessing cost, but maintain or even improve the original preprocessing cost.

III. MPC on Leveled Linear Secret Sharings

A. Technical Overview

Our contributions rely on the observation that for multiple MPC protocols, such as the BGW protocol [13], the three-party replicated secret sharing protocol of [21], or protocols based on masked secret sharing such as [27], [28], multiplications consist of two phases: In the first phase, the parties non-interactively obtain an alternative kind of secret sharing of the product. In the second phase, the parties interactively convert the result back to the protocol's original secret sharing scheme so that further multiplications are possible. For example, BGW multiplies two values, represented by polynomials f, g of maximum degree $d = t-1$ with party $\mathcal{P}_i$ holding shares $f(i), g(i)$, by each $\mathcal{P}_i$ setting $h(i) = f(i) \cdot g(i)$, defining a polynomial h that has double degree $2d$. In [21], two values shared in a replicated way can be non-interactively multiplied, obtaining an additive sharing of the product. In both cases, the second step upgrades the sharing back to the original sharing type, i.e., lowers the degree in the case of BGW or replicates all shares in [21], while also *re-randomizing* the sharing.

While the details for these specific protocols will be provided in the following subsections, we here aim to first establish the general motivation and build a generic concept: MPC with *leveled linear secret sharing*. Here, there are two internal linear secret sharing schemes, $\mathcal{S}$ and $\mathcal{S}'$, where multiplication of shares in $\mathcal{S}$ can be done non-interactively, yielding a share in $\mathcal{S}'$. This represents the first phase of the aforementioned multiplication protocols where $\mathcal{S}'$ then represents, e.g., Shamir shares [17] of degree $2d$ or additive shares, respectively. The threshold of $\mathcal{S}'$ needs to be at least that of $\mathcal{S}$ to preserve privacy, and we observe that in the above examples, it even becomes strictly larger. Furthermore, we allow non-interactively converting from $\mathcal{S}$ to $\mathcal{S}'$, which we will explain in detail later. This leads over to the following definition:

Definition 2 (Leveled Linear Secret Sharing (LLSS))**.** *A (t, n)-threshold* leveled linear secret sharing *(LLSS) scheme $(\mathcal{S}, \mathcal{S}')$ on domain $\mathcal{R}$ consists of:*

- *(t, n)-LinSSS $\mathcal{S}$ on $\mathcal{R}$. For ease of notation, we write $[\![v]\!] := [v]_{\mathcal{S}}$, and we call such sharings* higher sharings.
- *(t', n)-LinSSS $\mathcal{S}'$ on $\mathcal{R}$ for some $t' \geq t$. For ease of notation, we write $\langle v \rangle := [v]_{\mathcal{S}'}$, and we call such sharings* lower sharings.
- *A non-interactive procedure $\langle xy \rangle \leftarrow \mathsf{mult}([\![x]\!], [\![y]\!])$, multiplying two higher sharings, and yielding a lower sharing. This procedure may be augmented by also using preprocessing material which is obtained independently of $[\![x]\!], [\![y]\!]$, i.e., during the preprocessing phase of an MPC protocol. We only require the preprocessing for masked sharings in §III-E where we particularly optimize for online performance.*
- *A non-interactive procedure $\langle x \rangle \leftarrow \mathsf{downgrade}([\![x]\!])$ that converts from higher to lower sharing.*

The second step of multiplying in prior protocols in our case reflects an upgrade from $\mathcal{S}'$ to $\mathcal{S}$, which requires interaction:

$$[\![x]\!] \leftarrow \mathsf{upgrade}(\langle x \rangle).$$

More classical protocols, as in [13], [21], [27], [28], on a sharing scheme $\mathcal{S}$ reflect MPC on an LLSS where after each instance of mult, the result in $\mathcal{S}'$ is immediately upgraded back to $\mathcal{S}$ using $\mathsf{upgrade}$. Instead, we now allow to remain longer in $\mathcal{S}'$, making use of its linearity that still supports further linear operations. Before the next multiplication gates, we must ensure that an upgrade happens somewhere along the way so that all inputs to multiplications are always in $\mathcal{S}$. Furthermore, we also allow to provide inputs in and reconstruct outputs from the lower domain $\mathcal{S}'$. Hence, e.g., if multiple input values to a protocol are added and only the sum is later used in a multiplication, all of these inputs may be shared in $\mathcal{S}'$ if this is cheaper than inputs in $\mathcal{S}$, followed by a single upgrade on the sum. Similarly, between the last multiplication and an output, there is no need to upgrade to $\mathcal{S}$ before reconstructing.

Security: We provide the intuition regarding the security of our LLSS instantiations here and later provide the full, formal proofs in §B. For all prior protocols that we base our instantiations on, we observe that the security of their multiplications relies primarily on the security of the upgrading. For the first step, the non-interactive multiplication, it is only important that the result is a valid sharing of the expected product because only this is used as security argument regarding this step, and as no messages are exchanged. Hence, proceeding with linear operations between non-interactive multiplication and upgrade

preserves security, because these operations do not entail any communication but still result in a valid sharing of the expected value. Providing inputs directly in $\mathcal{S}'$ also is secure as per the properties of scheme $\mathcal{S}'$, which has a threshold at least that of $\mathcal{S}$. Regarding outputs directly from $\mathcal{S}'$, additional care is required. For example, in BGW [13], the output of a multiplication corresponds to a polynomial of degree $\leq 2d$ that is the product of two polynomials of degree $\leq d$. Yet, it is no random polynomial (up to representing the correct value) which was required earlier as an invariant, as it, e.g., cannot be reducible and of degree $> d$. This can lead to leakage upon further communication, depending on this polynomial [56]. Similarly, exchange of additive shares in [21] can disclose additional information as noted in [21]. Both approaches, therefore, integrate a re-randomization technique inside their upgrade. We additionally also integrate re-randomization in all outputs from $\mathcal{S}'$, ensuring that any output from $\mathcal{S}'$ can be simulated. Finally, the security of all linear operations, inputs, and outputs in $\mathcal{S}$ is inherited from the base protocols.

We proceed by introducing the notion of *extended circuits* that, while encoding the function to be computed, additionally also assign the choice of sharing semantics to all operations and specify where upgrades and downgrades happen (§III-B). Then, we generalize three LinSSS-based MPC protocols to LLSS (§III-C, §III-D, §III-E) and finally give an outlook on the possibility of using even more layers (§III-F).

B. Circuit Model

Usually, an arithmetic circuit $\mathcal{C}$ to be evaluated by an MPC protocol is a directed acyclic graph (DAG) where the nodes are multiplication, addition, and multiplication-by-constant gates. The edges are called wires, and multiplication and addition gates have exactly two input wires, while multiplication-by-constant gates have one input wire. Input and output gates are labeled with a party $\mathcal{P}_i$. Input gates have no input wires but output a value provided by $\mathcal{P}_i$. Symmetrically, output gates reveal the value on their input wire to $\mathcal{P}_i$.

To support the use of an LLSS, we define *extended circuits* $\mathcal{C}'$. In contrast to standard circuits $\mathcal{C}$, each wire now has an additional label $\mathcal{S}$ or $\mathcal{S}'$, indicating in which domain the value on this wire during circuit evaluation should be secret-shared. We additionally require that each addition or multiplication-by-constant gate uses the same domain for all of its input and output wires. Multiplication gates in accordance with Definition 2 require input wires to use $\mathcal{S}$ and output wires to use $\mathcal{S}'$. We furthermore introduce gates to downgrade and upgrade (cf. §III-A) where each downgrade gate has one input wire in $\mathcal{S}$ and output wires in $\mathcal{S}'$, and each upgrade has one input wire in $\mathcal{S}'$ and output wires in $\mathcal{S}$. For input and output gates, we allow their wires to use any domain if consistent, e.g., all output wires of an input gate must use the same domain. The domain of wires connected to input and output gates then specifies the domain in which the specific input or output is provided.

An extension $\mathcal{C}'$ of a circuit $\mathcal{C}$ is the result of adding sharing semantic labels to all wires of $\mathcal{C}$ and then adding downgrade and upgrade gates, replacing some wires by two wires, leading first from the source to a downgrade/upgrade and then to the destination, such that all rules specified above are satisfied.[2] Hence, $\mathcal{C}$ is $\mathcal{C}'$ after removing all sharing semantic labels and downgrade/upgrade gates. We refer to $\mathrm{Ext}(\mathcal{C})$ as the set of all possible extensions of $\mathcal{C}$.

C. Generalizing Replicated Secret Sharing

In the $(n = 3)$-party honest majority setting, the protocol of [21] based on replicated secret sharing [57] is a popular choice due to its high performance.[3] The secret sharing semantics are as follows:

Definition 3 (Three-Party Replicated Secret Sharing [57], [21], [22])**.** *Three-party* replicated secret sharing *(RSS) is a* $(2, 3)$*-threshold LinSSS over a ring* $\mathcal{R}$ *with* $v \in \mathcal{R}$ *shared as*

$$[v]^1_{RSS} := (v_1, v_3) \qquad [v]^2_{RSS} := (v_2, v_1) \qquad [v]^3_{RSS} := (v_3, v_2)$$

where $v_1, v_2, v_3 \leftarrow\!\!\$\ \mathcal{R}$ *subject to* $v_1 + v_2 + v_3 = v$.

For multiplication of $[x]_{\text{RSS}}, [y]_{\text{RSS}}$, [21], [22] exploit that

$$\begin{aligned} xy = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) = &(x_1y_1 + x_3y_1 + x_1y_3) + \\ &(x_2y_2 + x_2y_1 + x_1y_2) + \\ &(x_3y_3 + x_3y_2 + x_2y_3). \end{aligned}$$

The first summand can locally be computed by $\mathcal{P}_1$, the second by $\mathcal{P}_2$, and the third by $\mathcal{P}_3$, non-interactively yielding a $(3, 3)$-additive sharing (cf. Definition 1) or, in short, 3ASS-sharing of $z = xy$. Letting each party $\mathcal{P}_i$ then send its additive share to $\mathcal{P}_{i-1}$ would again yield an RSS-sharing of the product. Yet, for preserving privacy, it is necessary to first re-randomize $[z]_{\text{3ASS}}$ by adding a fresh random sharing $[0]_{\text{3ASS}}$, i.e., a 3ASS-sharing with shares independently and uniformly random subject to $[0]^1_{\text{3ASS}} + [0]^2_{\text{3ASS}} + [0]^3_{\text{3ASS}} = 0$. This sharing can be obtained using a protocol Π_{cr} for which [21] provides a non-interactive instantiation using pre-shared PRF keys.

We note that this process naturally consists of a non-interactive mult, yielding the product in 3ASS, and an interactive upgrade switching back to RSS. Hence, it induces an LLSS $(\text{RSS}, \text{3ASS})$ (cf. Definition 2) with higher sharings $[\![\cdot]\!] := [\cdot]_{\text{RSS}}$ and lower sharings $\langle\cdot\rangle := [\cdot]_{\text{3ASS}}$. Procedures mult and upgrade are given as parts of the multiplication protocol of [21], [22] depicted in Fig. 2. Downgrading a sharing $[\![v]\!]$ can be instantiated by each party $\mathcal{P}_i$ holding $[\![v]\!]^i = (v_i, v_{i-1})$ discarding v_{i-1} and setting $\langle v\rangle^i = v_i$.

Finally, recall from §III-A that for outputs in domain $\mathcal{S}' = \text{3ASS}$, we require re-randomization to ensure privacy. Concretely, as shown by [21], the outputs of mult are not random shares subject to representing the correct value, but an exchange of them may leak information. This is why the original multiplication protocol adds $\langle 0\rangle \leftarrow \Pi_{\text{cr}}()$ to the additive sharing before exchanging values. Similarly, we instantiate all output gates in $\mathcal{S}' = \text{3ASS}$ by computing a fresh $\langle 0\rangle \leftarrow \Pi_{\text{cr}}()$,

[2] If k wires with the same source require an upgrade, we also allow merging these upgrades by using a single wire to only one upgrade, followed by k wires from the upgrade to all destinations.

[3] In this work, we use the simplified formulation of the protocol from [22].

adding it to the sharing to be opened, and only then opening the sharing by sending all shares to the designated output party. For all other input and output gates, shares are simply exchanged as per the sharing schemes so that for inputs, all parties receive their shares, and for outputs, the output party receives sufficiently many shares to reconstruct the secret.

Protocol $[\![z]\!] \leftarrow \Pi_{\text{mult}}([\![x]\!], [\![y]\!])$

Input: $[\![\cdot]\!]$-sharings of $x, y \in \mathcal{R}$, *Output:* $[\![\cdot]\!]$-sharing of $z = x \cdot y$.

$\langle z \rangle = \mathsf{mult}([\![x]\!], [\![y]\!])$ (where $z = xy$)

- Each party $\mathcal{P}_i$ computes $\langle z \rangle^i = x_i y_i + x_{i-1} y_i + x_i y_{i-1}$.

$[\![z]\!] = \mathsf{upgrade}(\langle z \rangle)$

- The parties compute $\langle 0 \rangle \leftarrow \Pi_{\text{cr}}()$ and set $\langle z' \rangle \leftarrow \langle z \rangle + \langle 0 \rangle$.
- Each party $\mathcal{P}_i$ sends $\langle z' \rangle^i$ to $\mathcal{P}_{i+1}$ and receives $\langle z' \rangle^{i-1}$ from $\mathcal{P}_{i-1}$.
- Each party $\mathcal{P}_i$ sets $[\![z]\!]^i = (z_i, z_{i-1}) = (\langle z' \rangle^i, \langle z' \rangle^{i-1})$.

Figure 2: RSS-multiplication of [21], [22], split into non-interactive mult and interactive upgrade.

In §A-A, we provide an additional optimization for sharing protocol inputs using PRFs, and, for the sake of completeness, give a full, formal specification of our LLSS MPC protocol. Security is formally proven in §B, following the intuition provided in §III-A.

D. Generalizing Shamir's Secret Sharing

Next, we adapt Shamir's Secret Sharing Scheme [17] into an LLSS. Note that throughout this section, we require $\mathcal{R}$ to be a field with at least n distinct non-zero elements $\alpha_1, \ldots, \alpha_n$.

Definition 4 ((t,n)-Shamir's Secret Sharing [17])**.** *(t,n)-Shamir's secret sharing (tSSS) is a (t,n)-threshold LinSSS over a field $\mathcal{R}$ of size $> n$, splitting a secret $v \in \mathcal{R}$ into n values $[v]^i_{t\text{SSS}} \in \mathcal{R}$ where $(\alpha_i, [v]^i_{t\text{SSS}}) = (\alpha_i, f(\alpha_i))$ for $i \in \mathbb{N}_{\leq n}$ are points on a polynomial f over $\mathcal{R}$ with a maximum degree of $t-1$, and $f(0) = v$. Procedure* $\mathsf{share}(v)$ *chooses random $c_j \leftarrow\!\!\$\, \mathcal{R}$ for $j \in \mathbb{N}_{\leq t-1}$, sets $c_0 = v$, defines $f(X) := \sum_{j=0}^{t-1} c_j X^j$, and sets each share $[v]^i_{t\text{SSS}}$ to $f(i)$. Procedure* rec *takes at least t shares which are sufficient to compute f and then reconstructs $v = f(0)$.*

As introduced by the original BGW protocol [13], two secret-shared values $[x]_{t\text{SSS}}, [y]_{t\text{SSS}}$ are multiplied by each party first multiplying its two shares. Given that the shares of x respectively y correspond to polynomial $f(X)$ resp. $g(X)$ of maximum degree $t-1$, the products held by all parties now correspond to polynomial $h(X) = (f \cdot g)(X)$ representing $h(0) = f(0) \cdot g(0) = x \cdot y$. Yet, $h(X)$ has maximum degree $2t-2$, i.e., threshold $2t-1$, so that BGW requires $2t-1 \leq n$ for $h(X)$ to still be uniquely identified from the existing shares. This also is what requires the honest majority setting of BGW, setting $t = \lfloor (n+1)/2 \rfloor$. As the threshold would grow too large from further multiplications, there exist several methods to immediately decrease the polynomial degree back, e.g., [13], [58], [15], [16].

We again split these two parts. First, we consider $[\![\cdot]\!] := [\cdot]_{t\text{SSS}}$ as higher sharings. For the lower sharings, we use the results of the first multiplication step, i.e., local multiplication of individual shares. The resulting threshold of the multiplication is $2t-1 = n-1$ for even n and $2t-1 = n+1-1 = n$ otherwise. For the sake of consistency, we always interpret the result to have full threshold n, given that a polynomial of maximum degree $n-2$ also is a polynomial of maximum degree $n-1$.[4] Hence, for lower sharings we consider $\langle \cdot \rangle := [\cdot]_{n\text{SSS}}$ and consider an LLSS $(t\text{SSS}, n\text{SSS})$.

While our LLSS approach can use any degree reduction from [13], [58], [15], [16] for upgrading from $\langle \cdot \rangle$ to $[\![\cdot]\!]$, we follow the DN07 [15] approach for good scalability with n, also adding optimizations from [16]. The idea is to preprocess pairs $([\![r]\!], \langle r \rangle)$ for random $r \in \mathcal{R}$ and then, given some $\langle z \rangle$, compute $\langle z \rangle - \langle r \rangle$ and open this to one designated party $\mathcal{P}_{\text{king}}$.[5] This party can then distribute $z - r$ to all parties who finally set $[\![z]\!] = z - r + [\![r]\!]$ and the overall process just costs $\mathcal{O}(n)$ communication. For further optimization, we use the preprocessing from ATLAS [16] that requires $\mathcal{P}_{\text{king}}$ to $[\![\cdot]\!]$-share $z - r$ instead of distributing it as plaintext. This effectively halves the preprocessing cost while binding a specific party $\mathcal{P}_i$ to each individual multiplication to act as $\mathcal{P}_{\text{king}}$, hence using preprocessing material $([\![r]\!], \langle r \rangle, \mathcal{P}_i)$. The overall multiplication and our split of it into procedures mult and upgrade is given in Fig. 3.

Protocol $[\![z]\!] \leftarrow \Pi_{\text{mult}}([\![x]\!], [\![y]\!])$

Input: $[\![\cdot]\!]$-sharings of $x, y \in \mathcal{R}$, *Output:* $[\![\cdot]\!]$-sharing of $z = x \cdot y$.

$\langle z \rangle = \mathsf{mult}([\![x]\!], [\![y]\!])$ (where $z = xy$)

- Each party $\mathcal{P}_i$ computes $\langle z \rangle^i = [\![x]\!]^i [\![y]\!]^i$.

$[\![z]\!] = \mathsf{upgrade}(\langle z \rangle)$

- **Preprocessing:** Generate $([\![r]\!], \langle r \rangle, \mathcal{P}_{\text{king}})$ as described in [16].
- The parties compute $\langle z - r \rangle \leftarrow \langle z \rangle - \langle r \rangle$ and send their resulting shares to $\mathcal{P}_{\text{king}}$.
- $\mathcal{P}_{\text{king}}$ runs $(n\text{SSS}).\mathsf{rec}$ on the received shares to recover $z - r$, runs $(t\text{SSS}).\mathsf{share}(z - r)$, and sends the resulting shares of $[\![z - r]\!]$ to the corresponding parties.
- The parties compute $[\![z]\!] = [\![z - r]\!] + [\![r]\!]$.

Figure 3: DN07-style [15], [16] tSSS-multiplication, split into non-interactive mult and interactive upgrade.

For downgrading, we reinterpret a $[\![v]\!]$ as $\langle v \rangle$, each party using the same share as before. While this removes redundancy, it is obvious that the result still represents the same value. Finally, as noted in §III-A, outputs from domain $\mathcal{S}' = n\text{SSS}$ require additional re-randomization, which we achieve by adding a zero-sharing $\langle 0 \rangle$. Such zero-sharings can be generated during preprocessing, using the method described in [16]. We give the full formal protocol specification and additional optimizations using PRFs based on those by [16] in §A-B. and the full security proof in §B.

[4] The result technically still has threshold $n-1$, but we will later re-randomize to a proper threshold-n sharing before any communication so that privacy is preserved.

[5] The role of $\mathcal{P}_{\text{king}}$ rotates between parties across multiplications for load-balancing the communication.

E. Generalizing Masked Secret Sharing

In this section, we go beyond generalizing a specific MPC protocol to LLSS. Instead, we provide a general compiler that transforms any LinSSS and corresponding multiplication protocol into an LLSS-protocol. Our compiler utilizes and generically formalizes the masking circuit randomization technique as used in [26], [27], [28], leveraging function-dependent preprocessing to optimize online performance. We begin by providing an intuition for our construction, which is based on [26], [27], [28], and then give a full description.

Let there be a LinSSS $\mathcal{S}'$ (cf. §II-B) that we will use as lower sharing, i.e., $\langle\cdot\rangle := [\cdot]_{\mathcal{S}'}$. The idea is to define a *masked LinSSS* $\mathcal{S}$ as higher sharing $[\![\cdot]\!] := [\cdot]_{\mathcal{S}}$, where for secret $v \in \mathcal{R}$, a $\lambda_v \leftarrow\!\!\$\ \mathcal{R}$ is $\mathcal{S}'$-shared between the parties during preprocessing and additionally, all parties have the masked value $m_v = v - \lambda_v$ that is computed in the online phase. Hence, $[\![v]\!]^i = (m_v = v - \lambda_v, \langle\lambda_v\rangle^i)$ for each party $\mathcal{P}_i$. We note that by the linearity of $\mathcal{S}'$, $\mathcal{S}$ is linear too:

$$[\![cx + dy + e]\!] := (cm_x + dm_y + e, \langle c\lambda_x + d\lambda_y\rangle)$$

for $[\![x]\!], [\![y]\!]$ and public values $c, d, e \in \mathcal{R}$, as $m_{cx+dy+e} = cm_x + dm_y + e = c(x - \lambda_x) + d(y - \lambda_y) + e = cx + dy + e - c\lambda_x - d\lambda_y = cx + dy + e - \lambda_{cx+dy+e}$. For multiplication $z = x \cdot y$, we note that

$$\begin{aligned} z = xy &= (m_x + \lambda_x)(m_y + \lambda_y) \\ &= m_x m_y + m_x\lambda_y + m_y\lambda_x + \lambda_x\lambda_y. \end{aligned}$$

The parties precompute $\langle\lambda_x\lambda_y\rangle$, using a suitable multiplication protocol for $\mathcal{S}'$, that may be significantly more expensive than the multiplication in $\mathcal{S}$ that we aim to optimize as it runs in a less time-critical preprocessing phase. Then, in the online phase, the parties can compute

$$\langle z\rangle = m_x m_y + m_x\langle\lambda_y\rangle + m_y\langle\lambda_x\rangle + \langle\lambda_x\lambda_y\rangle$$

without interaction by the linearity of $\mathcal{S}'$. To upgrade $\langle z\rangle$ from $\mathcal{S}'$ to $\mathcal{S}$, during preprocessing, they establish a sharing $\langle\lambda_z\rangle$ for $\lambda_z \leftarrow\!\!\$\ \mathcal{R}$ and then compute $\langle m_z\rangle = \langle z\rangle - \langle\lambda_z\rangle$ online which they finally recover. Prior works use both steps together to implement their multiplication, while we again split them up. We proceed to formally specify our construction:

Theorem 1 (Masking Compilation of LinSSS to LLSS-MPC)**.** *Let $\mathcal{S}'$ be a (t, n)-LinSSS (cf. §II-B) and Π_{multPre} be a secure multiplication protocol on $\mathcal{S}'$ for up to $t-1$ corrupted parties. Then, there exists a masked version $M(\mathcal{S}')$ of $\mathcal{S}'$ which we use as higher sharing $\mathcal{S}$ s.t. $(\mathcal{S}, \mathcal{S}')$ is an LLSS and there exists an MPC protocol on this LLSS.*

Proof: The higher/masked sharing $\mathcal{S}$ is defined as $(m_v, \langle\lambda_v\rangle)$ for a secret $v \in \mathcal{R}$ where m_v is known to all parties and $v = m_v + \lambda_v$. For the sharing procedure $\mathcal{S}.\mathsf{share}(v)$, the parties establish $\langle\lambda_v\rangle$ for $\lambda_v \leftarrow\!\!\$\ \mathcal{R}$ during preprocessing so that the dealer learns λ_v.[6] For procedure $\mathcal{S}.\mathsf{rec}([\![v]\!])$, the parties run $\mathcal{S}'.\mathsf{rec}(\langle\lambda_v\rangle)$ towards the output receiver which then computes $v = m_v + \lambda_v$. The security with threshold t of $\mathcal{S}$ directly follows from that of $\mathcal{S}'$ with the same threshold. Linearity of $\mathcal{S}$ follows from that of $\mathcal{S}'$ as shown before.

For function-dependent preprocessing, the parties first for each upgrade from $\langle v\rangle$ to $[\![v]\!]$ establish $\langle\lambda_v\rangle$ for $\lambda_v \leftarrow\!\!\$\ \mathcal{R}$ which can be achieved non-interactively using PRFs [52], [59]. They also establish $\langle\lambda_v\rangle$ for all inputs in $\mathcal{S}$. For each multiplication $z = x \cdot y$, they then non-interactively compute $\langle\lambda_x\rangle, \langle\lambda_y\rangle$ following the circuit topology and linearity of $\mathcal{S}'$, given that $\langle\lambda_x\rangle, \langle\lambda_y\rangle$ are linear combinations on $\langle\lambda_v\rangle$-values sampled before for inputs in $\mathcal{S}$ and outputs of upgrades. Finally, the parties compute $\langle\lambda_x \cdot \lambda_y\rangle \leftarrow \Pi_{\mathrm{multPre}}(\langle\lambda_x\rangle, \langle\lambda_y\rangle)$ for all multiplications in parallel, yielding a constant-round preprocessing.

Multiplication now directly follows the prior intuition, and we provide the details in Fig. 4. Upgrading is defined in Fig. 5, supporting the optional use of the $\mathcal{P}_{\mathrm{king}}$-strategy [15] for many parties. For downgrading of $[\![v]\!]$, we simply let the parties compute $m_v + \langle\lambda_v\rangle$. Outputs from domain $\mathcal{S}'$ require additional re-randomization by adding a zero-sharing $\langle 0\rangle$. This can be generated non-interactively during preprocessing via pseudo-random zero sharing, using the techniques from [52].

Protocol $\langle z\rangle \leftarrow \mathsf{mult}([\![x]\!], [\![y]\!])$

Input: $[\![\cdot]\!]$-sharings of $x, y \in \mathcal{R}$, *Output:* $\langle\cdot\rangle$-sharing of $z = x \cdot y$.
Preprocessing Material: $\langle\lambda_x\lambda_y\rangle$.

- The parties compute $\langle z\rangle = m_x m_y + m_x\langle\lambda_y\rangle + m_y\langle\lambda_x\rangle + \langle\lambda_x\lambda_y\rangle$.

Figure 4: Masked, non-interactive multiplication.

Protocol $[\![z]\!] \leftarrow \mathsf{upgrade}(\langle z\rangle)$

Input: $\langle\cdot\rangle$-sharings of $z \in \mathcal{R}$, *Output:* $[\![\cdot]\!]$-sharing of z.
Preprocessing Material: $\langle\lambda_z\rangle$.

- The parties compute $\langle m_z\rangle = \langle z\rangle - \langle\lambda_z\rangle$.
- The parties reconstruct m_z using one of the following options:
  - A: Each party is sent enough shares of $\langle m_z\rangle$ to recover m_z using $\mathcal{S}'.\mathsf{rec}$.
  - B: A designated $\mathcal{P}_{\mathrm{king}}$ is sent enough shares of $\langle m_z\rangle$ to recover m_z using $\mathcal{S}'.\mathsf{rec}$ which it then sends to all other parties.
- Each party $\mathcal{P}_i$ sets $[\![z]\!]^i = (m_z, \langle\lambda_z\rangle^i)$.

Figure 5: Upgrading by masking.

■

Theorem 2. *The protocol from Theorem 1 is computationally secure against an adversary corrupting $< t$ parties.*

The intuition is as follows: The protocol is correct, and privacy follows from the fact that for upgrades, the exchanged data is masked by fresh pseudo-random shares $\langle\lambda_v\rangle^i$, while outputs from $\mathcal{S}$ use these pseudo-random masks, only revealing the output, and outputs from $\mathcal{S}'$ are re-randomized by addition of $\langle 0\rangle$. We give the full, formal proof in §B, and the complete protocol specification in §A-C.

Finally, we note that a protocol variant with a trusted dealer replacing Π_{multPre} exists, inspired by the design of [27]:

[6] E.g., the dealer could sample λ_v and $\mathcal{S}'.\mathsf{share}$ it between the parties.

Remark 3.1 (Masked LLSS-MPC with a Trusted Dealer)**.** *An MPC-protocol as in Theorem 1 can also be instantiated with an additional dealer party that does not have inputs or outputs and remains absent from the protocol's online phase. This dealer then samples all masks λ_v, $\langle\cdot\rangle$-shares them between the parties, and then computes and shares all $\langle\lambda_x\lambda_y\rangle$ required for the multiplications.*

1) Existing Concrete Instantiations: In the literature, there are multiple uses of MPC on masked secret sharings, but without using our idea of LLSS and instead running upgrade directly after each multiplication. For our evaluation (cf. §V-A), we consider the two-party (i.e., $t = 2$) protocol ABY2.0 [28]. This directly corresponds to using additive shares $\mathcal{S}' = 2\text{ASS}$ in our construction. Preprocessing multiplication Π_{multPre} is instantiated using oblivious transfer or homomorphic encryption. Another instantiation is ASTRA [27], which extends ABY2.0 with a dealer (cf. Remark 3.1). Here, random masks $\langle\lambda_v\rangle$ are set up non-interactively, sampling the individual shares from pre-shared PRF keys. The only interaction by the dealer is for each multiplication to $\langle\cdot\rangle$-share products $\lambda_x\lambda_y$. For active security (which is not our focus here), SWIFT [29] uses masking on $\mathcal{S}' = \text{RSS}$, Turbospeedz [26] uses additive shares with SPDZ-style MACs [19] as $\mathcal{S}'$, and Trident [60] and Asterisk [61] extend these approaches by a dealer.

2) Optimizing Preprocessing Communication: We note that in the prior protocols (§III-C, §III-D), mult did not require any communication, while here, it still requires interactive preprocessing. For example, computing a sum of products $\sum_{k=1}^{\ell}[\![x_k]\!][\![y_k]\!]$ in $\mathcal{S}$ could now be done with the online cost of a single upgrade after computing $\langle\sum_{k=1}^{\ell} x_k y_k\rangle$ without online interaction. Still, the preprocessing cost would scale with dimension ℓ as each of ℓ instances of mult requires to run Π_{multPre}. ASTRA [27], which supports dot-products with constant communication cost, instead lets its dealer not establish a sharing $\langle\lambda_{x_k}\lambda_{y_k}\rangle$ for each multiplication here, but instead has it distributing $\langle\sum_{k=1}^{\ell}\lambda_{x_k}\lambda_{y_k}\rangle$ directly as the individual parts do not matter for the final result. We can translate this approach to our generalized protocol as follows: For each instance of mult, we only compute $m_x m_y + m_x\langle\lambda_y\rangle + m_y\langle\lambda_x\rangle = \langle z - \lambda_x\lambda_y\rangle$ which does not require any preprocessing, but results in an error of $-\lambda_x\lambda_y$ in the result. Addition of $\langle\lambda_x\lambda_y\rangle$ would compensate this error, but requires preprocessing. Instead, we do not compensate such errors immediately. Then, as long as computation in $\mathcal{S}'$ continues without compensating errors, these errors will accumulate, noting that only linear operations are allowed. Somewhere before either outputting from $\mathcal{S}'$ or upgrading, we compensate the then accumulated error, e.g., in the prior example, by adding $\sum_{k=1}^{\ell}\lambda_{x_k}\lambda_{y_k}$. Hence, we effectively delay error compensation to an arbitrary position between multiplication(s) and the next following upgrade(s) or output(s) from $\mathcal{S}'$, aiming to later compensate less, accumulated errors.

Of course, even after moving the error compensation, the corresponding accumulated term still needs to be efficiently computed in preprocessing. If we have a dealer, this is easy as it holds all masks and, hence, can locally compute the compensation and secret-share it, similar to the idea in [27]. Otherwise, if the lower sharing $\mathcal{S}'$ is an upper sharing of another LLSS $(\mathcal{S}', \mathcal{S}'')$, we can recursively use our general idea of MPC on LLSS to reach constant communication per compensation to be computed. For each multiplication $z = x\cdot y$, we non-interactively multiply $\langle\lambda_x\rangle\cdot\langle\lambda_y\rangle$ to obtain $[\lambda_x\lambda_y]_{\mathcal{S}''}$ and proceed with linear operations in $\mathcal{S}''$ up to the point where the error compensation is required. Only then, the parties interact by upgrading the resulting error compensation from $\mathcal{S}''$ to $\mathcal{S}'$. The recursive approach works, e.g., for SWIFT [29] that uses the masked version of replicated secret sharing (Def. 3). Error compensations for individual multiplications can be computed non-interactively as additive sharings, accumulated, and then finally upgraded to replicated sharings as shown in §III-C. For ABY2.0 [28], this trick appears not to work as it seems there is no LLSS using additive shares as higher shares. In §C-E, we give details on our recursive optimization and show how it can be used to minimize preprocessing costs.

3) Relation to Beaver Multiplication [43]: Multiplication with Beaver triples [43] multiplies two shared values $[x], [y]$ by using a triple $([a], [b], [c])$ that is random subject to $c = ab$, reconstructing $x - a, y - b$ and then setting $[z] = [xy] = (x-a)(y-b) + (x-a)[b] + (y-b)[a] + [c]$. As pointed out in [28], this is equivalent to the masked approach here, reinterpreting a, b as masks λ_x, λ_y and $x-a, y-b$ as masked values m_x, m_y. The difference is that [43] essentially upgrades to masked sharings on the *input* wires of multiplications and immediately switches back to the original domain by multiplication, whereas, e.g., ABY2.0 [28] upgrades the *output* wires of multiplications instead so that all inputs to multiplication gates already are masked. This was motivated by multiplications having two inputs and one output, so that, per multiplication, fewer interactive upgrades are needed. Our generalized approach to masked secret sharing can, but does not necessarily, upgrade immediately before or after multiplications. Hence, it can emulate both Beaver multiplication [43] and ABY2.0 [28] at their respective original cost, but also more freely position upgrades at positions not directly adjacent to multiplication gates. This gives us unprecedented flexibility, filling the space between both prior approaches that randomize on inputs or outputs of multiplications, and, as we will see in §V-A, enable further optimization.

F. Multi-LLSS

While our MPC protocols for LLSS use two layers, where non-interactive multiplication has inputs on the higher and outputs on the lower layer, we raise the following question: *Could we build an LLSS with more than two layers with multiplications each going one layer down without online interaction, enabling multiple multiplications before needing to upgrade?* This appears to be possible! For instance, $M(\text{RSS})$ (masked sharing based on replicated sharings, cf. Theorem 1) as used by SWIFT [29] can be multiplied into RSS without online interaction (§III-E), and this can subsequently be multiplied into additive 3ASS without interaction (§III-C). This yields

two layers of multiplication without interaction. The same idea has recently been utilized by ALKAID [45], albeit only to hand-optimize AND-gates with three and four inputs, rather than introducing a leveled secret sharing for generic circuits. Similarly, if using BGW-style protocols (§III-D), we could require a more aggressive threshold, e.g., $t \approx n/4$, so that multiplications of tSSS yield $(2t-1)$SSS and multiplications of those yield $(4t-3)$SSS that can still be recovered by n parties. We could also imagine a masked version of tSSS, adding another layer on top. We leave further investigation of such multilayer schemes to future work. As a final remark, the masking compiler from §III-E may raise the idea of building an LLSS over arbitrarily many layers by recursively building masked sharings based on other masked sharings, e.g., using three layers $M(M(\mathcal{S}')), M(\mathcal{S}'), \mathcal{S}'$ for some LinSSS $\mathcal{S}'$. This unfortunately does not work as the first layer of multiplications would yield masks on the second layer that are unknown during preprocessing, preventing another non-interactive layer of multiplications. We give an example for this issue in §A-D.

IV. Sharing Assignment Optimization

Given a circuit $\mathcal{C}$ and an LLSS $(\mathcal{S}, \mathcal{S}')$, we want to find an extended circuit $\mathcal{C}'$ (cf. §III-B) which adds sharing semantic labels $\mathcal{S}$ and $\mathcal{S}'$ to each wire and augments the circuit by additional gates to downgrade from $\mathcal{S}$ to $\mathcal{S}'$ and upgrade from $\mathcal{S}'$ to $\mathcal{S}$, still computing the same function as $\mathcal{C}$.[7] Briefly recall from §III-B that we furthermore require linear gates to have the same sharing semantics on all inputs and outputs and that multiplications always have inputs in $\mathcal{S}$ and outputs in $\mathcal{S}'$. An obvious choice is to immediately upgrade the output of each multiplication from $\mathcal{S}'$ back to $\mathcal{S}$, keeping all linear operations, inputs, and outputs in $\mathcal{S}$. This exactly emulates existing MPC protocols such as 3PC on RSS [21], BGW [13], and ABY2.0 [28], using $\mathcal{S}'$ only as an intermediate step during multiplications (cf. Fig. 1a, p. 2). We have also seen in §III-E3 that upgrading to a masked sharing (cf. §III-E) on the *input* wires to each multiplication exactly emulates MPC with Beaver multiplication [43].

In cases where a few multiplication outputs (through linear gates) feed into many multiplication inputs, it is preferable to do the interactive upgrades early, as fewer wires need to be covered. In the opposite case, with many outputs feeding into a few inputs, it is preferable to upgrade late. Yet, it may also happen that linear arithmetic after a layer of multiplications first narrows to only a few wires, and then widens again into many multiplication inputs. In such a case, it may make sense to upgrade in-between linear, non-interactive operations to save on communication. We are interested in always using the choice of sharings on all wires and corresponding upgrades that optimizes communication, i.e., the best possible extension $\mathcal{C}'$ from the set of all possible extensions $\mathrm{Ext}(\mathcal{C})$ of a circuit $\mathcal{C}$, yielding the following optimization problem:

[7] We do not aim to also rewrite the actual arithmetics in the circuit for optimization. This would be logic minimization, an orthogonal problem which is Σ_2-hard [62].

Definition 5 (LLSS Sharing Assignment Problem)**.** *Let* COMM $: \{\mathcal{C}' \mid \mathcal{C}' \text{ extended circuit}\} \to \mathbb{R}_0$ *be a communication cost metric for extended circuits (this is specific to the used LLSS-protocol, and we may consider online or total communication). The* LLSS sharing assignment problem *is to, given a circuit $\mathcal{C}$, find an extended circuit $\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})$ with minimal* COMM*:*

$$\mathcal{C}^* = \operatorname*{arg\,min}_{\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})} \mathrm{COMM}(\mathcal{C}')$$

We note that the communication COMM computes the sum of required communication for each interactive gate in the circuit. In Tab. I, we provide the cost of all protocols from §III (with their PRF optimizations from §A), where $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}, \mathrm{C}_{\mathrm{in}}^{\mathcal{S}}$ denote the communication of sharing an input in lower respectively higher domain, $\mathrm{C}_{\mathsf{mult}}$ denotes the communication for a mult (which is non-interactive online, but still requires a static preprocessing cost in case of masked sharing), $\mathrm{C}_{\mathsf{up.}}$ denotes the communication of upgrade, $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'}, \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$ denote the communication of reconstructing from lower respectively higher domain towards one party, and $\mathrm{C}_{\$}$ denotes the communication for re-randomization which we require to do as part of each output from the lower domain (cf. §III-A).

Table I: Online communication costs for all gates except for free linear operations, for the protocols from §III. $|\mathcal{R}|$ is the size of one ring element, $t = \lfloor (n+1)/2 \rfloor$ is the threshold from §III-D, A is the underlying *masking scheme* from §III-E, and n is the number of parties. Preprocessing cost in footnotes.

	RSS (§III-C)	tSSS (§III-D)	MSS(A) (§III-E)
$\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$	0	0	$\mathrm{C}_{\mathrm{in}}^{A}$
$\mathrm{C}_{\mathrm{in}}^{\mathcal{S}}$	$\lvert\mathcal{R}\rvert$	$(n-t)\lvert\mathcal{R}\rvert$	$(n-1)\lvert\mathcal{R}\rvert$[a]
$\mathrm{C}_{\mathsf{mult}}$	0	0	0[b]
$\mathrm{C}_{\mathsf{up.}}$	$3\lvert\mathcal{R}\rvert$	$(2n-t-1)\lvert\mathcal{R}\rvert$[c]	$n\cdot\mathrm{C}_{\mathrm{out}}^{A}$ or $\mathrm{C}_{\mathrm{out}}^{A}+(n-1)\lvert\mathcal{R}\rvert$[d]
$\mathrm{C}_{\$}$	0	0	0
$\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'}$	$2\lvert\mathcal{R}\rvert$	$(n-1)\lvert\mathcal{R}\rvert$	$\mathrm{C}_{\mathrm{out}}^{A}$
$\mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$	$\lvert\mathcal{R}\rvert$	$(t-1)\lvert\mathcal{R}\rvert$	$\mathrm{C}_{\mathrm{out}}^{A}$

[a] prepr: establish random $[\lambda_v]_{\mathcal{S}'}$, λ_v known to dealer
[b] prepr: run Π_{multPre} [c] prepr: $\approx n\lvert\mathcal{R}\rvert/2$
[d] second option if $\mathcal{P}_{\mathrm{king}}$-strategy [15]

Given some circuit $\mathcal{C}$, we need to build a valid extension $\mathcal{C}'$ where the cost introduced by the positioning of upgrades and choice of domain for individual circuit inputs and outputs is minimized. We first look at multiplication gates, as these will inherently require their inputs to be in the higher domain $\mathcal{S}$ while providing their outputs in the lower domain $\mathcal{S}'$. In between multiplications that are connected by a path over linear gates, at least one upgrade needs to be placed so that the later multiplication receives its inputs in $\mathcal{S}$. In the resulting $\mathcal{C}'$, inputs to multiplications either are a direct result of an upgrade, or they are outputs of a linear gate that only receives inputs in $\mathcal{S}$ and, hence, fulfill the same property that their inputs come from upgrades or linear operations in $\mathcal{S}$. Thus, the sub-circuit between both multiplications needs to be split into a left-hand part that uses $\mathcal{S}'$, and a right-hand part that exclusively uses $\mathcal{S}$. The boundary between these parts is where upgrades from $\mathcal{S}'$ to $\mathcal{S}$ need to be placed, which are our main

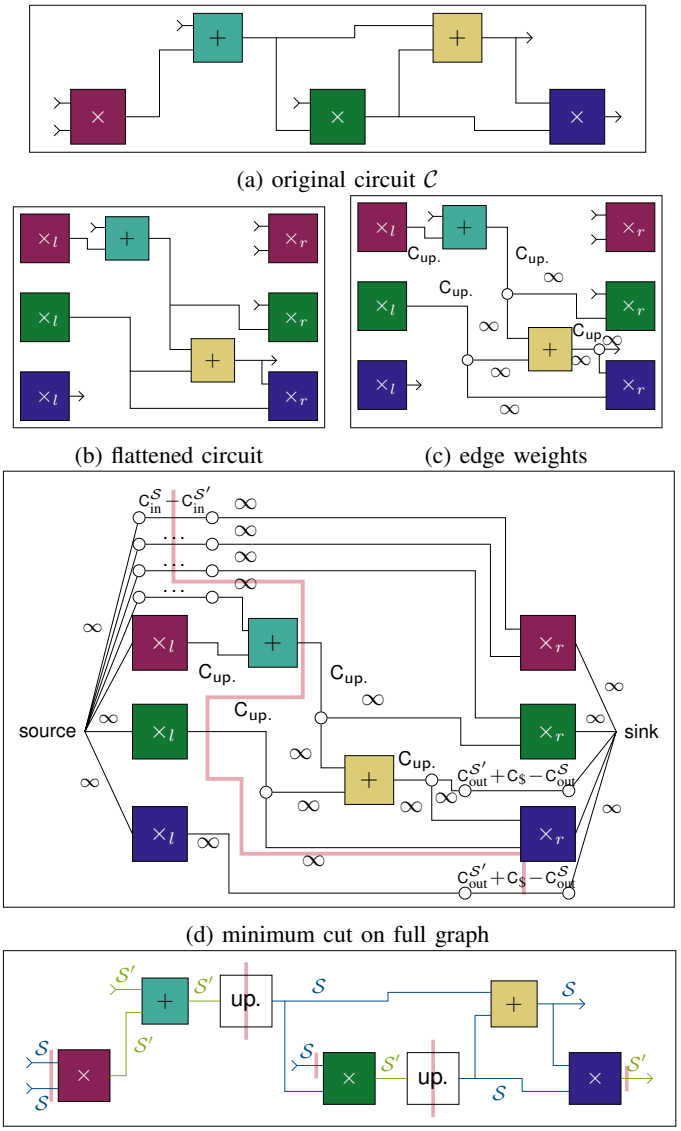
source of interaction cost. Hence, we aim to identify a possible split where the boundary contains the minimal number of wires. This corresponds to a minimum cut.

We now frame our optimization as a minimum cut problem. Each multiplication gate needs to act as a source, emitting $\mathcal{S}'$ wires, and as a sink, receiving $\mathcal{S}$-wires. Hence, we split each multiplication gate "$\times$" into two gates "$\times_l$" and "$\times_r$" where source "$\times_l$" has no inputs, but all original output wires of "$\times$", and sink "$\times_r$" has no outputs, but all original input wires of "$\times$". Between "$\times_l$" and "$\times_r$" of all multiplications, we place the remaining linear arithmetic of the circuit. The resulting *flattened circuit* for the example circuit $\mathcal{C}$ from Fig. 6a is given in Fig. 6b. Note that this construction also automatically takes care of paths over linear arithmetic that skip a multiplication layer, as visible in Fig. 6b.

The outputs of gates may be connected to multiple wires and, as defined in §III-B, a circuit extension allows to batch upgrades for multiple of them into a single upgrade, using one wire to the upgrade and from there, split into multiple wires to all destinations. As the number of possible sets of wires to augment by upgrades is exponential in the number of former output wires, it would be inefficient to enumerate them. Instead, we always upgrade all or no wires. If some wire later goes to a gate where it is required to use $\mathcal{S}'$, it is always possible to insert a downgrade gate which comes at no communication cost. Hence, in all cases where a gate output has multiple wires, we add an intermediate node, draw an edge from the gate output to this intermediate node, and then rewire all prior gate output wires to originate from the intermediate node instead.[8] This is depicted in Fig. 6c. At this point, we also add edge weights: Each output wire of a gate gets weight $\mathrm{C}_{\mathrm{up.}}$ as cutting this wire equals the insertion and, hence, cost of one upgrade. All outputs of intermediate nodes, as introduced before, have weight ∞ as we always want to cut before the node, i.e., upgrade the wire for all destinations.

Next, we need to take care of inputs and outputs to decide if they should be in the higher or lower domain.[9] We assume here that $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'} < \mathrm{C}_{\mathrm{in}}^{\mathcal{S}} < \mathrm{C}_{\mathrm{up.}} + \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$ and $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}} < \mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} < \mathrm{C}_{\mathrm{up.}} + \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$ which holds, e.g., for RSS and tSSS in Tab. I, as this is the most interesting case. Other cases are trivial: e.g., for $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'} \geq \mathrm{C}_{\mathrm{in}}^{\mathcal{S}}$, it is always optimal to input in $\mathcal{S}$ which, if needed, can be downgraded to $\mathcal{S}'$ for free at a total cost not higher than if the input was immediately in $\mathcal{S}'$. We provide an exhaustive list of trivial cases alongside their optimal solutions in §C-A and focus on the aforementioned difficult case here. Now, providing inputs in $\mathcal{S}'$ is cheaper than in $\mathcal{S}$, but might require additional upgrades later. For now, we assume that all inputs are in $\mathcal{S}'$. In that case, inputs are handled like multiplication outputs: paths starting at them need to be cut somewhere before reaching a multiplication input. For each input, we add a small gadget consisting of a source node and an edge to a second node, that can be cut to immediately upgrade the input. This second node is then connected to everything that the input is connected to in $\mathcal{C}$ by edges of weight ∞. Now, as $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} < \mathrm{C}_{\mathrm{up.}} + \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$, immediately providing the input in $\mathcal{S}$ is better than having it in $\mathcal{S}'$ and then upgrade. Hence, we set the weight of the gadget's internal edge to $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} - \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$, i.e., what one needs to pay additionally to input in $\mathcal{S}$ compared to $\mathcal{S}'$. Cutting this edge now represents that the input is in $\mathcal{S}$ instead of $\mathcal{S}'$, instead of representing an upgrade. This is also shown in our example in Fig. 6d.

For outputs, we use similar gadgets where the right node acts as a sink. Assume that all outputs are in $\mathcal{S}$ which is cheaper than outputs in $\mathcal{S}'$ by assumption. Then, wires using $\mathcal{S}'$ still require an upgrade, represented by cutting the gadget's edge. Now, if we allow outputs in either sharing,



(a) original circuit $\mathcal{C}$

(b) flattened circuit

(c) edge weights

(d) minimum cut on full graph

(e) optimal extended circuit $\mathcal{C}'$ (with annotated cut from Fig. 6d)

Figure 6: Example optimization from circuit $\mathcal{C}$ to best extended circuit $\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})$. Same coloring used for individual gates respectively the copies "$\times_l$", "$\times_r$" of "$\times$" gates.

[8] This roughly corresponds to Lawler's transformation [63] for hypergraphs.

[9] Orthogonally, [64] automatically finds opportunities for local computation on inputs known by the same party before input sharing. Both approaches could be combined.

it is cheaper to re-randomize and output in $\mathcal{S}'$ compared to upgrading and outputting in $\mathcal{S}$ as $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} < \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$. Hence, we set the edge weight to $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} - \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$, i.e., what one needs to pay additionally to output from $\mathcal{S}'$ (including re-randomization) compared to an output from $\mathcal{S}$. Cutting this edge then stands for having the output in $\mathcal{S}'$.

After introducing a single principal source and sink node to reduce the problem from the multi-source/sink case, we compute a minimum cut. This can be achieved by standard algorithms such as Edmonds-Karp [65] to compute the maximum flow and then searching for the boundary between the reachable and unreachable nodes in the residual graph of the flow, which yields the minimum cut [66]. In our example in Fig. 6d, the resulting minimum cut is shown. We translate that into an optimal extended circuit $\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})$ that is shown in Fig. 6e as follows: For each cut wire (outside of input/output gadgets), we add an upgrade gate. It may happen that there also exist wires traversing the cut backwards[10], in which case we add a free downgrade. Each cut edge in an input gadget (or its non-existence if $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} \leq \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$) indicates that the input is to be shared in $\mathcal{S}$ or immediately upgraded, whatever is cheaper; otherwise it will be in $\mathcal{S}'$. Similarly, each cut edge in an output gadget (or its non-existence if $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} \leq \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$) indicates that the output shall be in $\mathcal{S}'$, otherwise it is in $\mathcal{S}$. For conciseness, we defer a full, formal specification of our optimization algorithm to §C-B. To conclude, our optimization algorithm computes an optimal solution:

Theorem 3. *Our optimization algorithm (formal description in §C-B) solves the LLSS Sharing Assignment problem (Def. 5), i.e., given a circuit $\mathcal{C}$, computes an extended circuit $\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})$ with minimal communication cost* COMM.

This follows from the previous considerations of our construction and we provide a full proof in §C-C. The dominant part of the optimization cost is the Edmonds-Karp algorithm [65] with complexity $\mathcal{O}(|V||E|^2)$ for $|V|$ nodes and $|E|$ edges. As the number of gates (including inputs and outputs) is at most doubled in our construction and as the number of wires also gets at most doubled (by our merging of multiple output wires from the same gate), we get a complexity of $\mathcal{O}(|\mathcal{C}|^3)$. While we use a basic algorithm to compute the minimum cut for our share assignment, we note that more sophisticated methods exist with better complexity, see §C-D.

V. Evaluation

To analyze the communication improvement that our optimizer from §IV can find by using the additional freedom of detaching interactive upgrades from the remaining, now non-interactive multiplications (cf. §III), we implement our optimizer in C++ and present the results next. We compare to the prior approach of immediately upgrading after each multiplication as baseline. To also investigate the impact on concrete protocol run time, we implement our generalization of three-party MPC on replicated sharings from §III-C in C++, running on the extended circuits computed by our optimizer. We run benchmarks of this protocol on a server with an Intel Core i9-7960X CPU @2.80 GHz and 128 GB of RAM, and take the average run times over 10 iterations. To emulate realistic network behavior, we use the network emulation tool from NEON [67], considering a LAN setting with 1 Gbit/s bandwidth and 1 ms RTT and a WAN setting with 100 Mbit/s and 100 ms RTT. We make our implementations of optimizer and protocol publicly available at https://encrypto.de/code/LLSS-MPC.

We consider multiple practical circuits for our evaluation. Regarding binary circuits for 32-bit integer arithmetic, we use a ripple-carry adder (RCA), a parallel prefix adder (PPA)[11], and depth-optimized multiplication circuits from [32]. Furthermore, we benchmark binary circuits for single-precision floating point addition and multiplication from [32], and use the floating point division from [68]. We also consider circuits for symmetric cryptographic primitives, namely AES-128 (one block, including key expansion) and SHA-256 from [68], and LowMC-256 from [69]. While almost all existing benchmark circuit libraries for MPC are in the binary domain, we also benchmark on two custom arithmetic circuits on 64-bit values. One is for computing the mean square error (MSE) on a 768-dimensional vector. The second one is a neural network that operates on a 784-dimensional feature vector and has 3 hidden layers of sizes $(128, 128, 64)$ with a logistic activation function (approximated by a degree 6 polynomial) followed by 10 output nodes.[12] We assume the model weights to be public. Hence, multiplications are mainly used for the activation function. If not stated otherwise, we include the input and output phases for all circuits, where each output value is opened to a single fixed party.

Table II: Optimization times and memory utilization for our Sharing Assignment optimizer with the RSS-based cost model (Tab. I). Performance for other cost models similar with mostly $\pm 5\%$ run time and, in few cases $\pm 35\%$ run time. Measurements averaged over 5 runs.

Circuit	# of gates	Optimization Time	Memory (kB)
RCA	250	0.02 s	5060
PPA	397	0.04 s	5424
Multiplier	3064	1.11 s	14 372
float-add.	516	0.03 s	6020
float-mult.	2358	0.47 s	11 924
float-div.	184 199	145.54 min	626 048
AES-128	37 047	64.86 s	121 544
SHA-256	136 097	30.71 min	436 944
LowMC-256	266 205	11.25 min	848 840
MSE	3842	0.97 s	14 512
Neural-Net	254 800	4.42 min	632 356

Running our optimizer for a specific combination of protocol and circuit is a one-time expense, computing the optimal

[10] These will not be part of the cut, but go to gates left of the cut.

[11] PPAs are a very popular choice in secret-sharing based MPC to achieve a low round complexity for, e.g., conversions [1], [28], [32], which are crucial for efficient privacy-preserving machine learning.

[12] Fixed-point numbers can be used here. With low depth and for simplicity, we use no truncation here, but our protocols can be adapted with e.g. probabilistic truncation [70].

Table III: Total online communication in bits for various circuits using different baseline protocols that immediately upgrade after multiplication, and our LLSS-protocol extensions of them.

Circuit	**Replicated:** 3PC based on RSS			**Shamir:** 10PC based on tSSS			**Masked:** 2PC based on $M(2\text{ASS})$		
	base. ([21])	**ours** (§III-C)	**impr.**	base. ([16])	**ours** (§III-D)	**impr.**	base. ([28], [27])	**ours** (§III-E)	**impr.**
RCA	189	185	2.11%	3276	3208	2.07%	158	154	2.53%
PPA	573	358	37.52%	9932	6172	37.85%	414	252	39.13%
Multiplier	3450	3057	11.39%	59 800	52 952	11.45%	2332	2052	12.00%
float-add.	507	386	23.86%	8852	6744	23.81%	354	268	24.29%
float-mult.	2541	2236	12.00%	44 108	38 812	12.00%	1710	1500	12.28%
float-div.	246 999	184 627	25.25%	4 281 316	3 200 116	25.25%	164 730	123 106	25.26%
AES-128	19 584	13 232	32.43%	339 456	229 184	32.48%	13 184	8864	32.76%
SHA-256	68 743	67 080	2.41%	1 191 204	1 162 372	2.42%	46 170	45 055	2.45%
LowMC-256	5884	5590	4.99%	102 224	96 932	5.17%	4120	3717	9.78%
MSE	196 736	49 344	74.91%	836 160	197 440	76.38%	147 584	49 280	66.60%
Neural-Net	296 576	247 040	16.70%	1 268 864	1 070 720	15.61%	214 656	164 480	23.37%

extended circuit and then using this circuit for many protocol executions. Hence, we can tolerate slower optimization to decrease the communication and run time of all subsequent evaluations of a circuit. Still, as shown in Tab. II, for smaller circuits, our optimization runs in at most a couple of seconds. For the largest benchmarked circuits, our Sharing Assignment optimizer takes less than three hours as a one-time cost which therefore still is practical. For even larger circuits, we give possible directions for algorithm improvements in §C-D.

Next, we inspect improvements in communication (§V-A), in run time (§V-C), and give further evaluation results in §D.

A. Communication Improvements

We use our optimizer from §IV for our LLSS-protocols from §III-C (based on 3PC on RSS [21]), §III-D (based on BGW-style [13] MPC using Shamir's Secret Sharing [17] tSSS), and §III-E for two parties and underlying LinSSS 2ASS ($M(2\text{ASS})$ which corresponds to ABY2.0 [28] or, if using a dealer, ASTRA [27]). Here, we decide to optimize for online communication to streamline the computation once the inputs are provided. Note that the protocol from §III-C has no preprocessing, and the protocol from §III-D has the majority of its communication online while our optimization will also improve preprocessing. The preprocessing of the protocol from §III-E can be optimized in presence of a dealer (cf. §III-E2) and remains unchanged for two parties without a dealer. We provide the concrete preprocessing results in §D-A and focus on online performance here. For the protocol from §III-D, which works for an arbitrarily many parties $n > 2$, we choose $n = 10$ here and show in §D-B that the results for $n = 3$ are similar. For the protocols from §III-C and §III-E, we use $\mathcal{R} = \mathbb{F}_2$ as binary domain and $\mathcal{R} = \mathbb{Z}_{2^{64}}$ as arithmetic domain. As the protocol from §III-D requires fields larger than the number of parties, we then use $\mathbb{F}_{2^4}$ respectively $\mathbb{F}_p$ for a 64-bit prime p.

Our results are shown in Tab. III. We observe that on binary circuits, our improvement ranges from 2.07% up to 39.13% with the arithmetic operations (except for the higher depth RCA) and AES all having improvement above 10%. This result is quite notable because these improvements do not stem from specific new protocols, but rather from a generic construction that yields these improvements on top of multiple existing MPC protocols. Comparing the three different settings, we observe that they feature similar trends, whereas the existing differences can be explained by the relation between upgrading, input, and output cost differing across protocols. Regarding the arithmetic circuits MSE and Neural-Net, we see substantial improvement, especially for MSE with $> 66\%$.

B. Discussion: LLSS vs Dot Product Gates

We note that substantial improvement for MSE stems from this circuit essentially computing one big dot product. Similar improvements can be achieved by simply using a more specialized dot product gate [41], [40], [28]. Yet, in general, our general LLSS approach has multiple advantages over the use of dot product gates. First, no specialized gate for dot products and protocol implementation is required anymore to get communication improvements, and we do not require the circuit designer to identify and handle dot products as a special case. More importantly, our optimizer finds dot-product-like structures automatically. Hence, it also exploits such structures for circuits that are not consciously designed to contain a dot product such as support vector machines, but feature similar structures incidentally such as in a PPA. As an example, dot products are usually not actively considered on a high level for the design of binary circuitry, and yet, our optimizer finds significant improvement without the need to manually identify which parts of a circuit could be rewritten to dot products. Finally, our optimizer also finds further optimizations, e.g., by choosing the optimal domain for each input and output. The exact effect of that is discussed in more detail in §D-C.

C. Run Time Improvements

Next, we investigate the run time impact of our optimization in the case of our RSS-based protocol from §III-C, compared to the baseline protocol from [21]. We consider the batched parallel execution of 1000 copies of a circuit, given that especially most of the binary circuits are relatively small on their own, so that computation and communication would be negligibly small. The results are shown in Tab. IV.

For LAN, we observe at least 9% better run time for all but two circuits. It varies whether run time or communication improvement (cf. Tab. III) is higher, given different circuit topologies where communication can have a varying impact

Table IV: Run time (seconds) for the baseline RSS-based protocol from [21] that immediately upgrades after multiplication, and our LLSS-protocol extension from §III-C. Benchmarks using batch size 1000.

Circuit	LAN			WAN		
	base. ([21])	**ours** (§III-C)	**impr.**	base. ([21])	**ours** (§III-C)	**impr.**
RCA	0.04	0.04	2.51%	2.77	2.64	4.44%
PPA	0.03	0.02	13.33%	0.99	0.91	8.77%
Multiplier	0.05	0.04	9.23%	1.82	1.72	5.38%
float-add.	0.04	0.03	13.83%	2.02	1.99	1.53%
float-mult.	0.06	0.04	25.67%	2.32	2.29	1.20%
float-div.	5.40	5.30	1.79%	277.93	277.04	0.31%
AES-128	0.23	0.20	11.54%	5.58	5.40	3.32%
SHA-256	2.69	2.44	9.29%	123.37	123.27	0.08%
LowMC-256	0.74	0.63	15.70%	2.50	2.30	7.81%
MSE	0.38	0.17	54.64%	18.71	1.83	90.21%
Neural-Net	3.57	2.23	37.60%	15.64	6.57	57.98%

on run time, besides other factors such as local computation and round complexity. In the WAN setting, the improvement is mostly significantly lower than in the LAN setting. Notable exceptions are MSE and Neural-NET, where improvement now even reaches 90% and 58%, respectively. This can be attributed to both circuits having high communication (cf. Tab. III) and few rounds. We note that our optimization aims to minimize communication, not the number of rounds. It can save one round by not requiring interactive upgrades after the last multiplication layer but output directly from $\mathcal{S}'$ instead of $\mathcal{S}$, which is particularly beneficial for shallow circuits. However, further savings are not possible with our techniques, as between each pair of successive multiplications (that are the interactive gates in more classical protocols, but not for us), an interactive upgrade is required. In a WAN, the increased RTT now has a significantly greater impact, reducing our improvement for somewhat deeper circuits while even assisting to increase it for very shallow circuits.

VI. Conclusion and Future Work

In this work, we have introduced MPC on *leveled linear secret sharings* (LLSS), a new paradigm that we have shown to bring improvements to multiple well-established MPC protocols like three-party computation on replicated sharings [21], the BGW protocol [13], and protocols based on masked sharings such as ABY2.0 [28] and ASTRA [27]. The prior works follow a "multiply-then-reshare/upgrade" approach, two steps that we separate from each other, gaining new flexibility. Our novel, efficient optimizer optimally utilizes this flexibility for any circuit, decreasing communication and run time. We have also introduced the generalized notion of *masked secret sharing* to a generic compiler to create MPC protocols with highly efficient online phases, which is of independent interest.

For future work, we highlight three possible directions. First, a generalization to more than two layers as briefly addressed in §III-F provides potential for even more optimization, also regarding round complexity. Second, constructions based on leveled homomorphic encryption [71] may be rephrased in our framework, generalizing it beyond classical MPC and using our optimizer in other domains. Finally, MPC based on LLSS could be extended to the active security setting using, e.g., zero-knowledge fully linear interactive oracle proofs (zk-FLIOPs) [72], [73], and we defer further technical details to §E.

Acknowledgments

The authors thank Jan Filipp (TU Darmstadt) for his LowMC [69] Bristol-fashion circuit implementation.

This project received funding from the ERC under the EU's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/ 850990). It was co-funded by the DFG within SFB 1119 CROSSING/236615297, and supported by BMFTR and HMWK within ATHENE.

References

[1] P. Mohassel and P. Rindal, "ABY^3: A Mixed Protocol Framework for Machine Learning," in *ACM CCS*, 2018.

[2] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications," in *ACM ASIACCS*, 2018.

[3] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE S&P*, 2017.

[4] K. Gupta, N. Jawalkar, A. Mukherjee, N. Chandran, D. Gupta, A. Panwar, and R. Sharma, "SIGMA: Secure GPT Inference with Function Secret Sharing," *PoPETS*, 2024.

[5] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma, "Orca: FSS-based Secure Training and Inference with GPUs," in *IEEE S&P*, 2024.

[6] Q. Pang, J. Zhu, H. Möllering, W. Zheng, and T. Schneider, "BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers," in *IEEE S&P*, 2024.

[7] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure Graph Analysis at Scale," in *ACM CCS*, 2021.

[8] J. Yu, K. Chen, Y. Chen, X. Fan, X. Zhu, C. Hong, and W. Chen, "GraphAce: Secure Two-Party Graph Analysis Achieving Communication Efficiency," in *USENIX Security*, 2022.

[9] N. Koti, V. B. Kukkala, A. Patra, and B. R. Gopal, "Graphiti: Secure Graph Computation Made More Scalable," in *ACM CCS*, 2024.

[10] A. Brüggemann, N. Koti, V. B. Kukkala, and T. Schneider, "MultiCent: Secure and Scalable Computation of Centrality Measures on Multilayer Graphs," *PoPETS*, 2025.

[11] E. Baum, S. Buxbaum, N. Mathai, M. Faisal, V. Kalavri, M. Varia, and J. Liagouris, "ORQ: Complex Analytics on Private Data with Strong Security Guarantees," in *Symposium on Operating Systems Principles*, 2025.

[12] P. Mohassel, P. Rindal, and M. Rosulek, "Fast Database Joins and PSI for Secret Shared Data," in *ACM CCS*, 2020.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)," in *ACM STOC*, 1988.

[14] D. Chaum, C. Crépeau, and I. Damgård, "Multiparty Unconditionally Secure Protocols (Extended Abstract)," in *ACM STOC*, 1988.

[15] I. Damgård and J. B. Nielsen, "Scalable and Unconditionally Secure Multiparty Computation," in *CRYPTO*, 2007.

[16] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song, "ATLAS: Efficient and Scalable MPC in the Honest Majority Setting," in *CRYPTO*, 2021.

[17] A. Shamir, "How to Share a Secret," *Communications of the ACM*, vol. 22, no. 11, 1979.

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

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

[20] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority," in *CRYPTO*, 2018.

[21] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority," in *ACM CCS*, 2016.

[22] Y. Lindell and A. Nof, "A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority," in *ACM CCS*, 2017.

[23] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs," in *ACM CCS*, 2019.

[24] A. P. K. Dalskov, D. Escudero, and M. Keller, "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security," in *USENIX Security*, 2021.

[25] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority," in *EUROCRYPT*, 2017.

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

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

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

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

[30] A. C.-C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *IEEE FOCS*, 1986.

[31] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS*, 2015.

[32] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION – A Framework for Mixed-Protocol Multi-Party Computation," *ACM TOPS*, 2022.

[33] R. Garg, K. Yang, J. Katz, and X. Wang, "Scalable Mixed-Mode MPC," in *IEEE S&P*, 2024.

[34] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits," in *CRYPTO*, 2020.

[35] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic Protocol Selection in Secure Two-Party Computations," in *ACNS*, 2014.

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

[37] M. Ishaq, A. L. Milanova, and V. Zikas, "Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing," in *ACM CCS*, 2019.

[38] V. Fang, L. Brown, W. Lin, W. Zheng, A. Panda, and R. A. Popa, "CostCO: An Automatic Cost Modeling Framework for Secure Multi-Party Computation," in *IEEE Euro S&P*, 2022.

[39] E. Chen, J. Zhu, A. Ozdemir, R. S. Wahby, F. Brown, and W. Zheng, "Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols," in *IEEE S&P*, 2023.

[40] O. Catrina and S. de Hoogh, "Improved Primitives for Secure Multiparty Integer Computation," in *Security and Cryptography for Networks*, 2010.

[41] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast Large-Scale Honest-Majority MPC for Malicious Adversaries," in *CRYPTO*, 2018.

[42] A. Brüggemann, T. Schneider, and M. Stillger, "Additions, Multiplications, and the Interaction In-Between: Optimizing MPC Protocols via Leveled Linear Secret Sharing," in *NDSS*, 2027.

[43] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *CRYPTO*, 1991.

[44] H. Morita, E. Pohle, K. Sadakane, P. Scholl, K. Tozawa, and D. Tschudi, "MAESTRO: Multi-Party AES Using Lookup Tables," in *USENIX Security*, 2025.

[45] Y. Dong, X. Chen, X. Song, Y. Yang, W.-J. Lu, T. Zhang, J. Zhou, and J.-S. Dong, "ALKAID: Accelerating Three-Party Boolean Circuits by Mixing Correlations and Redundancy," *IEEE TIFS*, vol. 21, 2026.

[46] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," in *Theory of Cryptography*, 2005.

[47] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption Without Bootstrapping," in *ITCS*, 2012.

[48] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *CRYPTO*, 2013.

[49] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for Automating Secure Two-partY computations," in *ACM CCS*, 2010.

[50] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning," in *IEEE Euro S&P*, 2019.

[51] N. Gilboa and Y. Ishai, "Compressing Cryptographic Resources," in *CRYPTO*, 1999.

[52] R. Cramer, I. Damgård, and Y. Ishai, "Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation," in *Theory of Cryptography*, 2005.

[53] O. Goldreich, *Foundations of Cryptography: Basic Applications*. Cambridge University Press, 2004, vol. 2.

[54] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *Journal of Cryptology*, vol. 13, no. 1, 2000.

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

[56] G. Asharov and Y. Lindell, "A Full Proof of the BGW Protocol for Perfectly Secure Multiparty Computation," *Journal of Cryptology*, vol. 30, no. 1, 2017.

[57] M. Ito, A. Saito, and T. Nishizeki, "Secret Sharing Scheme Realizing General Access Structure," in *IEEE Global Telecommunication Conference (Globecom)*, 1987. [Online]. Available: https://archiv.infsec.ethz.ch/education/as09/secsem/papers/ItSaNi87.pdf

[58] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography," in *Principles of Distributed Computing*, 1998.

[59] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz, "Efficient Multi-party Computation over Rings," in *EUROCRYPT*, 2003.

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

[61] B. Karmakar, N. Koti, A. Patra, S. Patranabis, P. Paul, and D. Ravi, "Asterisk: Super-fast MPC with a Friend," in *IEEE S&P*, 2024.

[62] D. Buchfuhrer and C. Umans, "The Complexity of Boolean Formula Minimization," in *Automata, Languages and Programming*, 2008.

[63] E. L. Lawler, "Cutsets and partitions of hypergraphs," *Networks*, vol. 3, no. 3, 1973.

[64] F. Kerschbaum, "Automatically Optimizing Secure Computation," in *ACM CCS*, 2011.

[65] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM*, vol. 19, no. 2, 1972.

[66] L. R. Ford Jr and D. R. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics*, vol. 8, 1956.

[67] A. Klinger, V. Ehrmanntraut, and U. Meyer, "Estimating the Runtime and Global Network Traffic of SMPC Protocols," in *ACM CODASPY*, 2024.

[68] D. Archer, V. A. Abril, S. Lu, P. Maene, N. Mertens, D. Sijacic, and N. Smart, "'Bristol Fashion' MPC Circuits," https://nigelsmart.github.io/MPC-Circuits/, 2019.

[69] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *EUROCRYPT*, 2015.

[70] M. Zbudila, E. Pohle, A. Abidin, and B. Preneel, "MaSTer: Maliciously Secure Truncation for Replicated Secret Sharing Without Pre-Processing," in *CANS*, 2024.

[71] A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N. P. Smart, "Between a Rock and a Hard Place: Interpolating between MPC and FHE," in *ASIACRYPT*, 2013.

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

[73] A. Brüggemann, A. Nof, and T. Schneider, "One Proof to Rule Them All: Practical, Sublinear Verification for Actively Secure MPC on $\mathbb{Z}_{2^k}$ with Dishonest Majority and a Dealer," in *ACM CCS*, 2026.

[74] I. L. Beckenbach, "Matchings and Flows in Hypergraphs," Dissertation, Freie Universität Berlin, 2019. [Online]. Available: http://dx.doi.org/10.17169/refubium-2157

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

[76] A. Brüggemann, O. Schick, T. Schneider, A. Suresh, and H. Yalame, "Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater," in *IEEE S&P*, 2024.
[77] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, and B. Pinkas, "High-Throughput Secure AES Computation," in *ACM WAHC@CCS*, 2018.
[78] R. W. F. Lai, G. Malavolta, and D. Schröder, "Homomorphic Secret Sharing for Low Degree Polynomials," in *ASIACRYPT*, 2018.

APPENDIX A
PROTOCOL DETAILS

A. Details on LLSS-Protocol Based on Replicated Secret Sharing from §III-C

1) Full Protocol Specification: We provide the full specification of our protocol from §III-C in Fig. 7.

Protocol $\Pi_{\text{MPC}(\text{RSS},\text{3ASS})}$

Input: Each party $\mathcal{P}_i$ provides a vector of input values $\vec{x}_i \in \mathcal{R}^*$.

Auxiliary Public Input: Extended circuit $\mathcal{C}'$.

Output: Each party $\mathcal{P}_i$ receives a vector of output values $\vec{y}_i \in \mathcal{R}^*$ where $(\vec{y}_1, \ldots, \vec{y}_n) = \mathcal{C}'(\vec{x}_1, \ldots, \vec{x}_n)$.

Online:
All parties iterate over the gates of $\mathcal{C}'$ in the same topological order.

1. For every input gate labeled with dealer party $\mathcal{P}_D$ and with output wires in semantic $s \in \{\text{RSS}, \text{3ASS}\}$, $\mathcal{P}_D$ computes $s.\mathsf{share}(x_{D,k})$ on its corresponding input $x_{D,k}$ and sends each share to the corresponding party. The parties write their shares to all output wires of the input gate.

2. For every gate g that is no input or output gate with input wire(s) holding sharings $[v_1]_s$ and $[v_2]_s$ or constant $c \in \mathcal{R}$ if g is an addition or multiplication by constant, compute and write the following to the output wires:
   - if $g = v_1 + v_2$: $[v_1 + v_2]_s \leftarrow [v_1]_s + [v_2]_s$.
   - if $g = v_1 + c$: $[v_1 + c]_s \leftarrow [v_1]_s + c$.
   - if $g = c \cdot v_1$: $[cv_1]_s \leftarrow c[v_1]_s$.
   - if $g = v_1 \cdot v_2$ (requires that $s = \text{RSS}$): every party $\mathcal{P}_i$ parses $[v_1]^i_{\text{RSS}} = (a_1, b_1), [v_2]^i_{\text{RSS}} = (a_2, b_2)$ and sets $[v_1 \cdot v_2]^i_{\text{3ASS}} \leftarrow (a_1 b_1 + a_2 b_1 + a_1 b_2)$.
   - if $g = \mathsf{downgrade}(v_1)$ (requires that $s = \text{RSS}$): every party $\mathcal{P}_i$ parses $[v_1]^i_{\text{3ASS}} = (a, b)$ and sets $[v_1]^i_{\text{3ASS}} \leftarrow a$.
   - if $g = \mathsf{upgrade}(v_1)$ (requires that $s = \text{3ASS}$):
     - The parties compute $[0]_{\text{3ASS}} \leftarrow \Pi_{\text{cr}}()$ [21] and set $[v'_1]_{\text{3ASS}} \leftarrow [v_1]_{\text{3ASS}} + [0]_{\text{3ASS}}$.
     - Each party $\mathcal{P}_i$ sends $[v'_1]^i_{\text{3ASS}}$ to $\mathcal{P}_{i+1}$ and receives $[v'_1]^{i-1}_{\text{3ASS}}$ from $\mathcal{P}_{i-1}$.
     - Each party $\mathcal{P}_i$ computes $[v_1]^i_{\text{RSS}} \leftarrow ([v'_1]^i_{\text{3ASS}}, [v'_1]^{i-1}_{\text{3ASS}})$.

3. For every output gate labeled with receiver party $\mathcal{P}_R$ and with input wire in semantic $s \in \{\text{RSS}, \text{3ASS}\}$ holding $[v]_s$, $\mathcal{P}_R$ expects an output $y_{R,k}$ which is obtained as follows:
   - if $s = \text{RSS}$: Party $\mathcal{P}_{R-1}$ sends the second component of its share $[v]^{R-1}_{\text{RSS}}$ to $\mathcal{P}_R$ which can then locally derive full $[v]^{R-1}_{\text{RSS}}$.
   - if $s = \text{3ASS}$:
     - The parties compute $[0]_{\text{3ASS}} \leftarrow \Pi_{\text{cr}}()$ [21] and set $[v']_{\text{3ASS}} \leftarrow [v]_{\text{3ASS}} + [0]_{\text{3ASS}}$.
     - The parties send their shares of $[v']_{\text{3ASS}}$ to $\mathcal{P}_R$.
   - In any case, $\mathcal{P}_R$ computes $y_{R,k}$ by running $s.\mathsf{rec}$ on its own and the received shares.

Figure 7: Three-party MPC protocol on LLSS $(\text{RSS}, \text{3ASS})$ from §III-C.

2) Optimizing Inputs with PRFs: As noted in §II-B, pseudorandom secret sharing (PRSS) [52] provides an opportunity to decrease communication for sharing inputs by utilizing PRFs. Here, we first note that a 3ASS-sharing can be obtained non-interactively by the input party $\mathcal{P}_i$ generating share $\langle v \rangle^j$ together with each $\mathcal{P}_j$, using pre-shared keys instead of needing to send the value. It can then set $\langle v \rangle^i$ accordingly to v minus the sum of all other shares. For inputs in RSS, input party $\mathcal{P}_i$ can, together with the other parties, non-interactively sample $[\![v]\!]^{i+1}$, together with only $\mathcal{P}_{i-1}$ sample $[\![v]\!]^{i-1}$, and then compute from that and v the remaining value $[\![v]\!]^i$ that it needs to finally send to $\mathcal{P}_{i+1}$.

B. Details on LLSS-Protocol Based on Shamir's Secret Sharing from §III-D

1) Full Protocol Specification: We provide the full specification of our protocol from §III-D in Fig. 8.

Protocol $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$

Input: Each party $\mathcal{P}_i$ provides a vector of input values $\vec{x}_i \in \mathcal{R}^*$.

Auxiliary Public Input: Extended circuit $\mathcal{C}'$.

Output: Each party $\mathcal{P}_i$ receives a vector of output values $\vec{y}_i \in \mathcal{R}^*$ where $(\vec{y}_1, \ldots, \vec{y}_n) = \mathcal{C}'(\vec{x}_1, \ldots, \vec{x}_n)$.

Preprocessing:

- Generate one triple $([r]_{t\text{SSS}}, [r]_{n\text{SSS}}, \mathcal{P}_{\text{king}})$ for each $\mathsf{upgrade}$ gate, as described in [16].
- Generate fresh sharing $[0]_{n\text{SSS}}$ for each output from nSSS, as described in [16].

Online:
All parties iterate over the gates of $\mathcal{C}'$ in the same topological order.

1. For every input gate labeled with dealer party $\mathcal{P}_D$ and with output wires in semantic $s \in \{t\text{SSS}, n\text{SSS}\}$, $\mathcal{P}_D$ computes $s.\mathsf{share}(x_{D,k})$ on its corresponding input $x_{D,k}$ and sends each share to the corresponding party. The parties write their shares to all output wires of the input gate.

2. For every gate g that is no input or output gate with input wire(s) holding sharings $[v_1]_s$ and $[v_2]_s$ or constant $c \in \mathcal{R}$ if g is an addition or multiplication by constant, compute and write the following to the output wires:
   - if $g = v_1 + v_2$: $[v_1 + v_2]_s \leftarrow [v_1]_s + [v_2]_s$.
   - if $g = v_1 + c$: $[v_1 + c]_s \leftarrow [v_1]_s + c$.
   - if $g = c \cdot v_1$: $[cv_1]_s \leftarrow c[v_1]_s$.
   - if $g = v_1 \cdot v_2$ (requires that $s = t\text{SSS}$): every party $\mathcal{P}_i$ sets $[v_1 \cdot v_2]^i_{n\text{SSS}} \leftarrow [v_1]^i_{t\text{SSS}} \cdot [v_2]^i_{t\text{SSS}}$.
   - if $g = \mathsf{downgrade}(v_1)$ (requires that $s = t\text{SSS}$): every party $\mathcal{P}_i$ sets $[v_1]^i_{n\text{SSS}} \leftarrow [v_1]^i_{t\text{SSS}}$.
   - if $g = \mathsf{upgrade}(v_1)$ (requires that $s = n\text{SSS}$):
     - The parties consume one triple $([r]_{t\text{SSS}}, [r]_{n\text{SSS}}, \mathcal{P}_{\text{king}})$ from preprocessing, first setting $[v_1 - r]_{n\text{SSS}} \leftarrow [v_1]_{n\text{SSS}} - [r]_{n\text{SSS}}$ and sending their resulting shares to $\mathcal{P}_{\text{king}}$.
     - $\mathcal{P}_{\text{king}}$ runs $(n\text{SSS}).\mathsf{rec}$ on the received shares to recover $v_1 - r$.
     - $\mathcal{P}_{\text{king}}$ runs $(t\text{SSS}).\mathsf{share}(v_1 - r)$ and sends each share to the corresponding party.
     - The parties compute $[v_1]_{t\text{SSS}} \leftarrow [v_1 - r]_{t\text{SSS}} + [r]_{t\text{SSS}}$.

3. For every output gate labeled with receiver party $\mathcal{P}_R$ and with input wire in semantic $s \in \{t\text{SSS}, n\text{SSS}\}$ holding $[v]_s$, $\mathcal{P}_R$ expects an output $y_{R,k}$ which is obtained as follows:
   - if $s = t\text{SSS}$: $t - 1$ parties send their shares $[v]^i_{t\text{SSS}}$ to $\mathcal{P}_R$ which can then locally derive full $[v]^{R-1}_{\text{RSS}}$.
   - if $s = n\text{SSS}$:
     - The parties consume one $[0]_{n\text{SSS}}$ from preprocessing and set $[v']_{n\text{SSS}} \leftarrow [v]_{n\text{SSS}} + [0]_{n\text{SSS}}$.
     - The parties send their shares $[v']_{n\text{SSS}}$ to $\mathcal{P}_R$.
   - In any case, $\mathcal{P}_R$ computes $y_{R,k}$ by running $s.\mathsf{rec}$ on its own and the received shares.

Figure 8: n-party MPC protocol on LLSS $(t\text{SSS}, n\text{SSS})$ from §III-D.

2) Optimizing Inputs and Preprocessing with PRFs: If computational security is sufficient, PRFs with pre-shared keys can be used to reduce the communication for sharing inputs, which in turn also optimizes the upgrades' preprocessing and online phases, as described in [16]. In our case, we observe that to establish a $[\cdot]_{t'\text{SSS}}$-sharing for any $t' \in \mathbb{N}_{\leq n}$, only $n - t'$ ring elements need to be sent: The dealer, instead of choosing random coefficients for the underlying polynomial, chooses $t'-1$ shares non-interactively with $t'-1$ other parties, using the PRF keys. These shares and the secret to be shared fully determine the polynomial, and the dealer computes all remaining shares (including its own), sending them to the remaining $n - t'$ parties. Note that this decreases the cost for inputs in $\mathcal{S} = t\text{SSS}$ and, following that, also the online cost of $\mathsf{upgrade}$ where $\mathcal{P}_{\text{king}}$ secret-shares a value. Inputs in $\mathcal{S}' = n\text{SSS}$ are even non-interactive.

During preprocessing, different parties $\mathcal{P}_i$ initially provide pairs $(\llbracket r^i \rrbracket, \langle r^i \rangle)$ for $r^i \leftarrow\!\!\$\ \mathcal{R}$. Given one such pair from each of the n parties, [15] extracts $n - t + 1$ pairs where to the adversary, their secrets appear independently and uniformly random. In [16], $t - 1$ of these pairs are then again expanded into n triples of format $(\llbracket r \rrbracket, \langle r \rangle, \mathcal{P}_{\text{king}})$, one for each choice of $\mathcal{P}_{\text{king}}$. Hence, for the preprocessing of each $\mathsf{upgrade}$, we need $\frac{(t-1)\cdot n/(n-t+1)}{n}$ initial pairs $(\llbracket r^i \rrbracket, \langle r^i \rangle)$. Using the PRF optimization, each such pair costs only $n-t$ messages for $\llbracket r^i \rrbracket$ and none for $\langle r^i \rangle$ in [16]. We go one step further, noting that as $r^i \leftarrow\!\!\$\ \mathcal{R}$, we can choose t shares non-interactively that define the entire polynomial, including r^i, instead of choosing $t-1$ many and separately sampling random r^i. This saves one additional message. The amortized cost per $\mathsf{upgrade}$ preprocessing then finally consists of $\frac{(t-1)\cdot n/(n-t+1)}{n} \cdot (n - t - 1) = \frac{(t-1)(n-t-1)}{n-t+1}$ ring elements.

C. Full Specification for the Generic Masking-Based LLSS-Protocol from §III-E

We provide the full specification of our protocol from §III-E in Fig. 8. The protocol is generically compiled from a LinSSS $\mathcal{S}'$ and multiplication protocol Π_{multPre} on $\mathcal{S}'$ and inherits the number of parties, corruption threshold, and security from $\mathcal{S}'$ and Π_{multPre}.

Protocol $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$

Input: Each party $\mathcal{P}_i$ provides a vector of input values $\vec{x}_i \in \mathcal{R}^*$.

Auxiliary Public Input: Extended circuit $\mathcal{C}'$.

Output: Each party $\mathcal{P}_i$ receives a vector of output values $\vec{y}_i \in \mathcal{R}^*$ where $(\vec{y}_1, \ldots, \vec{y}_n) = \mathcal{C}'(\vec{x}_1, \ldots, \vec{x}_n)$.

Function-Dependent Preprocessing:

1. For each $\mathsf{upgrade}$ gate, generate $\langle \lambda_v \rangle$ for $\lambda_v \leftarrow\!\!\$\ \mathcal{R}$ which can be achieved non-interactively using PRFs [52], [59] or any equivalent technique specific to $\mathcal{S}'$. Write the resulting mask to the output wires.

2. For every input gate labeled with dealer party $\mathcal{P}_D$ and with output wires in semantic $M(\mathcal{S}')$, $\mathcal{P}_D$ computes $\mathcal{S}'.\mathsf{share}(\lambda_v)$ on $\lambda_v \leftarrow\!\!\$\ \mathcal{R}$ and sends each share to the corresponding party, or the parties use any equivalent technique specific to $\mathcal{S}'$ to compute random $[\lambda_v]_{\mathcal{S}'}$ so that $\mathcal{P}_D$ learns λ_v. Write the resulting mask to the output wires.

3. All parties iterate over the gates of $\mathcal{C}'$ in the same topological order. For every gate g that is no input or output gate, with inputs and outputs in $M(\mathcal{S}')$ and with input wire(s) holding mask sharings $[\lambda_{v_1}]_{\mathcal{S}'}$ and $[\lambda_{v_2}]_{\mathcal{S}'}$ or constant $c \in \mathcal{R}$ if g is an addition or multiplication by constant, compute and write the following masks to the output wires:

- if $g = v_1 + v_2$: $[\lambda_{v_1+v_2}]_{\mathcal{S}'} \leftarrow [\lambda_{v_1}]_{\mathcal{S}'} + [\lambda_{v_2}]_{\mathcal{S}'}$.
- if $g = v_1 + c$: $[\lambda_{v_1+c}]_{\mathcal{S}'} \leftarrow [\lambda_{v_1}]_{\mathcal{S}'}$.
- if $g = c \cdot v_1$: $[\lambda_{cv_1}]_{\mathcal{S}'} \leftarrow c \cdot [\lambda_{v_1}]_{\mathcal{S}'}$.
- ignore all other gates.

4. For each mult gate with input wires holding mask sharings $[\lambda_{v_1}]_{\mathcal{S}'}$ and $[\lambda_{v_2}]_{\mathcal{S}'}$, the parties interactively compute $[\lambda_{v_1}\lambda_{v_2}]_{\mathcal{S}'} \leftarrow \Pi_{\text{multPre}}([\lambda_{v_1}]_{\mathcal{S}'}, [\lambda_{v_2}]_{\mathcal{S}'})$.

5. Generate fresh sharing $[0]_{\mathcal{S}'}$ for each output from $\mathcal{S}'$, using pseudo-random zero sharing with the techniques from [52], or any equivalent technique specific to $\mathcal{S}'$.

Online:
All parties iterate over the gates of $\mathcal{C}'$ in the same topological order.

1. For every input gate labeled with dealer party $\mathcal{P}_D$ and with output wires in semantic $s \in \{M(\mathcal{S}'), \mathcal{S}'\}$, $\mathcal{P}_D$ computes $s.\mathsf{share}(x_{D,k})$ on its corresponding input $x_{D,k}$ (if $s = M(\mathcal{S}')$, using the $[\lambda_{x_{D,k}}]_{\mathcal{S}'}$ computed in preprocessing) and sends the resulting $m_{x_{D,k}}$ or $\mathcal{S}'$-shares to all parties. The parties write their shares to all output wires of the input gate.

2. For every gate g that is no input or output gate with input wire(s) holding sharings $[v_1]_s$ and $[v_2]_s$ or constant $c \in \mathcal{R}$ if g is an addition or multiplication by constant, compute and write the following to the output wires:

- if $g = v_1 + v_2$: $[v_1 + v_2]_s \leftarrow [v_1]_s + [v_2]_s$.
- if $g = v_1 + c$: $[v_1 + c]_s \leftarrow [v_1]_s + c$.
- if $g = c \cdot v_1$: $[cv_1]_s \leftarrow c[v_1]_s$.
- if $g = v_1 \cdot v_2$ (requires that $s = M(\mathcal{S}')$): $[v_1 \cdot v_2]_{\mathcal{S}'} \leftarrow m_{v_1}m_{v_2} + m_{v_1}[\lambda_{v_2}]_{\mathcal{S}'} + m_{v_2}[\lambda_{v_1}]_{\mathcal{S}'} + [\lambda_{v_1}\lambda_{v_2}]_{\mathcal{S}'}$ (using the $[\lambda_{v_1}\lambda_{v_2}]_{\mathcal{S}'}$ computed in preprocessing).
- if $g = \mathsf{downgrade}(v_1)$ (requires that $s = M(\mathcal{S}')$): $[v_1]_{\mathcal{S}'} \leftarrow [\lambda_{v_1}]_{\mathcal{S}'} + m_{v_1}$.
- if $g = \mathsf{upgrade}(v_1)$ (requires that $s = \mathcal{S}'$):
  - The parties compute $[m_{v_1}]_{\mathcal{S}'} \leftarrow [v_1]_{\mathcal{S}'} - [\lambda_{v_1}]_{\mathcal{S}'}$ (using the $[\lambda_{v_1}]_{\mathcal{S}'}$ computed in preprocessing).
  - The parties reconstruct m_{v_1} using one of the following options:
    - A: Each party is sent enough shares of $[m_{v_1}]_{\mathcal{S}'}$ to recover m_{v_1} using $\mathcal{S}'.\mathsf{rec}$.
    - B: A designated $\mathcal{P}_{\text{king}}$ is sent enough shares of $[m_{v_1}]_{\mathcal{S}'}$ to recover m_{v_1} using $\mathcal{S}'.\mathsf{rec}$ which it then sends to all other parties.
  - Each party $\mathcal{P}_i$ sets $[v_1]_{M(\mathcal{S}')} \leftarrow (m_{v_1}, [\lambda_{v_1}]^i_{\mathcal{S}'})$.

3. For every output gate labeled with receiver party $\mathcal{P}_R$ and with input wire in semantic $s \in \{M(\mathcal{S}'), \mathcal{S}'\}$ holding $[v]_s$, $\mathcal{P}_R$ expects an output $y_{R,k}$ which is obtained as follows:

- if $s = M(\mathcal{S}')$:
  - $t - 1$ parties send their shares $[\lambda_v]^i_{\mathcal{S}'}$ to $\mathcal{P}_R$ which then reconstructs λ_v using $\mathcal{S}'.\mathsf{rec}$.
  - $\mathcal{P}_R$ computes $y_{R,k} \leftarrow m_v + \lambda_v$.
- if $s = \mathcal{S}'$:
  - The parties consume one $[0]_{\mathcal{S}'}$ from preprocessing and set $[v']_{\mathcal{S}'} \leftarrow [v]_{\mathcal{S}'} + [0]_{\mathcal{S}'}$.
  - $t - 1$ parties send their shares $[v']^i_{\mathcal{S}'}$ to $\mathcal{P}_R$ which then reconstructs $y_{R,k}$ using $\mathcal{S}'.\mathsf{rec}$.

Figure 9: Generic MPC protocol on LLSS $(M(\mathcal{S}'), \mathcal{S}')$ from §III-E.

D. Why Building Multi-LLSS (§III-F) from Recursively Masking Does Not Work

We here provide a counter-example to recursively applying the masking compiler from §III-E to build a multi-LLSS as proposed in §III-F. Assume any LinSSS $\mathcal{S}'$. We now consider layers $M(M(\mathcal{S}'))$, $M(\mathcal{S}')$, $\mathcal{S}'$, following the masking technique from §III-E. Note that $\mathcal{S}'$ could also be a masked sharing scheme and that there may be more layers below $\mathcal{S}'$, but we will concentrate on the given three layers where problems will already occur.

We now aim to multiply two top layer sharings $[a]_{M(M(\mathcal{S}'))} = (m_a, [\lambda_a]_{M(\mathcal{S}')})$ and $[b]_{M(M(\mathcal{S}'))} = (m_b, [\lambda_b]_{M(\mathcal{S}')})$. To this end, the parties non-interactively compute

$$[c]_{M(\mathcal{S}')} = m_a m_b + m_a[\lambda_b]_{M(\mathcal{S}')} + m_b[\lambda_a]_{M(\mathcal{S}')} + [\lambda_a\lambda_b]_{M(\mathcal{S}')}.$$

Share $[\lambda_a\lambda_b]_{M(\mathcal{S}')}$ is computed during preprocessing. Noting that we also recursively have $[\lambda_a]_{M(\mathcal{S}')} = (m_{\lambda_a}, [\lambda_{\lambda_a}]_{\mathcal{S}'})$, $[\lambda_b]_{M(\mathcal{S}')} = (m_{\lambda_b}, [\lambda_{\lambda_b}]_{\mathcal{S}'})$, and $[\lambda_a\lambda_b]_{M(\mathcal{S}')} = (m_{\lambda_a\lambda_b}, [\lambda_{\lambda_a\lambda_b}]_{\mathcal{S}'})$, it holds that $[c]_{M(\mathcal{S}')} = (m_c, [\lambda_c]_{\mathcal{S}'})$ where

$$\begin{aligned} m_c &= m_a m_b + \quad m_a m_{\lambda_b} + \quad m_b m_{\lambda_a} + \quad m_{\lambda_a\lambda_b}, \\ [\lambda_c]_{\mathcal{S}'} &= \quad m_a[\lambda_{\lambda_b}]_{\mathcal{S}'} + \quad m_b[\lambda_{\lambda_a}]_{\mathcal{S}'} + \quad [\lambda_{\lambda_a\lambda_b}]_{\mathcal{S}'}. \end{aligned}$$

The issue arises if we aim to non-interactively multiply $[c]_{M(\mathcal{S}')}$ by some $[d]_{M(\mathcal{S}')} = (m_d, [\lambda_d]_{\mathcal{S}'})$, moving down one further layer from $M(\mathcal{S}')$ to $\mathcal{S}'$. The parties need to compute

$$[e]_{\mathcal{S}'} = m_c m_d + m_c[\lambda_d]_{\mathcal{S}'} + m_d[\lambda_c]_{\mathcal{S}'} + [\lambda_c\lambda_d]_{\mathcal{S}'}.$$

To obtain $[\lambda_c\lambda_d]_{\mathcal{S}'}$, $[\lambda_c]_{\mathcal{S}'}$ and $[\lambda_d]_{\mathcal{S}'}$ need to be multiplied. The idea for MPC based on masked secret sharing is to do this during function-dependent preprocessing so that the online multiplication remains non-interactive. Yet, it now holds that $\lambda_c = m_a\lambda_{\lambda_b} + m_b\lambda_{\lambda_a} + \lambda_{\lambda_a\lambda_b}$, depending on m_a, m_b which are computed online, given values a, b and preprocessing values λ_a, λ_b. Hence, computation of $[\lambda_c\lambda_d]_{\mathcal{S}'}$ does not exclusively depend on information known during preprocessing, and therefore, it needs to be computed online. Then, the second multiplication $[c]_{M(\mathcal{S}')} \cdot [d]_{M(\mathcal{S}')} = [e]_{\mathcal{S}'}$ requires online communication, contradicting the central objective of LLSS to make multiplications (without upgrades) non-interactive.

APPENDIX B
SECURITY PROOFS FOR OUR NOVEL LLSS-BASED MPC PROTOCOLS

We define our ideal functionality $\mathcal{F}_{\text{MPC}}$ for computing a function defined by some Boolean/arithmetic (extended) circuit $\mathcal{C}$ in Fig. 10. Note that we consider standard circuits and extended circuits as defined in §III-B here, as the latter do not change arithmetics, but only secret sharing semantics for our optimization.

Functionality $\mathcal{F}_{\text{MPC}}$

Input: Each party $\mathcal{P}_i$ inputs $\vec{x^i}$.

Public Parameter: (Extended) circuit $\mathcal{C}$.

Output: Each party $\mathcal{P}_i$ receives $\vec{y^i}$ where $(\vec{y^i})_{i\in\mathbb{N}_{\leq n}} = \mathcal{C}((\vec{x^i})_{i\in\mathbb{N}_{\leq n}})$.

Figure 10: Ideal functionality for securely computing an (extended) circuit $\mathcal{C}$ with n parties where each party provides some subset of the input-wire values and receives some subset of the output-wire values.

Security of an n-party protocol Π to compute an ideal functionality $\mathcal{F}$ is defined as existence of a probabilistic, efficient simulator S such that for each subset $\mathcal{P}_I$ of parties, $I \subseteq \mathbb{N}_{\leq n}$ with size up to a certain corruption threshold depending on the concrete protocol's setting, and all input vectors $\vec{x}$,

$$\{(\mathsf{S}^I(1^\kappa, \vec{x}_I, \mathcal{F}(\vec{x})_I), \mathcal{F}(\vec{x}))\} \equiv_c \{(\mathsf{view}_I^\Pi(\vec{x}, \kappa), \mathsf{out}^\Pi(\vec{x}, \kappa))\}.$$

Here, $\vec{x}_I$ denotes the inputs of the corrupted parties in $\mathcal{P}_I$, $\mathcal{F}(\vec{x})_I$ denotes their outputs from the ideal functionality, view_I^Π contains the corrupted parties' random tapes, inputs, and received messages from the protocol execution, out^Π denotes the outputs of all parties in the protocol execution, and $\equiv_c$ is computational indistinguishability w.r.t. security parameter κ.

We show security for our protocols from §III-C, §III-D, and §III-E.

Theorem 4. *The three-party protocol $\Pi_{\text{MPC(RSS,3ASS)}}$ from §III-C (fully specified in Fig. 7) is computationally secure against a passive adversary corrupting at most one party.*

Theorem 5. *The n-party protocol $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$ from §III-D (fully specified in Fig. 8) is unconditionally[13] secure against a passive adversary corrupting at most a minority of $t-1$ parties.*

Theorem 6 (Theorem 2 Restated)**.** *Let $\mathcal{S}'$ be a (t,n)-LinSSS and Π_{multPre} be a secure multiplication protocol on $\mathcal{S}'$ (for up to $t-1$ corrupted parties). The n-party protocol $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$ from §III-E (fully specified in Fig. 9) is computationally secure against a passive adversary corrupting at most a minority of $t-1$ parties.*

Due to similarities and resulting overlaps in the individual security proofs, we provide a single, general proof that makes case distinctions where needed.

Proof: We first observe that all our MPC protocols are correct, i.e., for each computed gate, the sharing on its output wire(s) represents the value resulting from the underlying operation, using the values that are represented by the sharings on the input wire(s) as inputs. For upgrade and downgrade gates, it is easy to see that the output value is the same as the input value while only the used sharing semantic is changed. We also note that the functionality $\mathcal{F}_{\text{MPC}}$ is deterministic.

We proceed by providing a simulator S which, given the identities of corrupt parties, their inputs, and their outputs,

[13] or computationally, if PRF optimizations as in §A-B2 are utilized

computes a transcript/view for the corrupt parties that cannot be distinguished from the real-world transcript/view that adversary $\mathcal{A}$ would learn from a protocol execution. The simulator begins by following the protocol specification on behalf of all parties, using the actual inputs of all corrupt parties and 0 for each input of an honest party.

Preprocessing: For the preprocessing (where applicable), it is easy to see that simulated and real transcripts so far are indistinguishable, given that this is independent of any inputs and instead only runs on freshly sampled random data. We additionally observe the following: For all triples $([r]_{t\text{SSS}}, [r]_{n\text{SSS}}, \mathcal{P}_{\text{king}})$ in $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$ where $\mathcal{P}_{\text{king}}$ is corrupt, their random values $r \in \mathcal{R}$ are uniformly and independently random and unknown to $\mathcal{A}$. This follows from their generation in [16] that creates batches of n triples where each party once is $\mathcal{P}_{\text{king}}$, and each subset of at most $t-1$ triples has independently random values r. Furthermore, in $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$, values λ_v except for those corresponding to inputs of a corrupted party remain hidden from $\mathcal{A}$, given that the only interaction for computation on these values is via secure Π_{multPre}. The same holds for the individual shares held by the honest parties.

Inputs: In the online phase, the only differences between simulation and real world are the 0-inputs by honest parties. Yet, the transcripts of the corrupt parties stay indistinguishable as per the t-privacy of the sharing procedure of the used LinSSSs. In particular, it is well established that t-privacy (or even privacy to a higher threshold) is provided by $\text{RSS}, \text{3ASS}, t\text{SSS}, n\text{SSS}$. It is also provided for $\mathcal{S}'$ for $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$ per assumption, and each input in $M(\mathcal{S}')$ transmits a value to all parties that is masked by random λ_v from preprocessing.

Circuit Evaluation: Further interaction before the output stage now only happens for each upgrade. In $\Pi_{\text{MPC}(\text{RSS},\text{3ASS})}$, the corrupt party $\mathcal{P}_i$ receives one value $[v'_1]^{i-1}_{\text{3ASS}}$. This is masked by random $[0]^{i-1}_{\text{3ASS}}$ and, hence, simulated and real-world transcript cannot be distinguished. In $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$, a corrupt $\mathcal{P}_{\text{king}}$ receives $[v_1]^i_{n\text{SSS}} - [r]^i_{n\text{SSS}}$ from each party $\mathcal{P}_i$. All individual shares are independently and uniformly random as per the aforementioned generation of $[r]_{n\text{SSS}}$, yielding a random reconstructed value $v_1 - r$ and, hence, indistinguishability between simulation and real world. A corrupt party that is not serving as $\mathcal{P}_{\text{king}}$ receives the result of $(t\text{SSS}).\text{share}$, yielding indistinguishability as per the privacy of the LinSSS. Finally, $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$ has each party $\mathcal{P}_i$ compute $[v_1]^i_{\mathcal{S}'} - [\lambda_{v_1}]^i_{\mathcal{S}'}$ where the second value acts as a mask that—for honest parties—is not known to $\mathcal{A}$. $\mathcal{A}$ learns at most all individual masked parts (if it is $\mathcal{P}_{\text{king}}$ or no $\mathcal{P}_{\text{king}}$ is used, otherwise it learns less, i.e., the sum over all these terms). Due to the masking, indistinguishability between simulation and real world is maintained.

Outputs: Most complex is the final part of the protocol where the honest parties reveal outputs to the corrupt parties (for outputs to honest parties, $\mathcal{A}$ does not receive any message). In these cases, the simulator must behave differently from the protocol. There are two cases for outputting, based on whether the outputs are revealed in the higher domain or the lower domain.

Lower Domain: The simpler case is outputting from the lower domain, where we re-randomize before exchanging shares. For an expected output v which the simulator learns as part of its input, it first uses the $s.\text{rec}$ of the corresponding LinSSS s to recover the actual current value $\hat{v}$ in the simulation. Then, the simulator computes $s.\text{share}(v-\hat{v})$ and uses the result of that instead of re-randomizer $[0]_s$. In $\Pi_{\text{MPC}(\text{RSS},\text{3ASS})}$, it simulates $\Pi_{\text{cr}}()$ returning $[v-\hat{v}]_{\text{3ASS}}$. In $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$, it simulates that the preprocessing returns $[v-\hat{v}]_{n\text{SSS}}$ instead of $[0]_{n\text{SSS}}$ using the corresponding protocol from [16]. This requires changing the preprocessing, which already happened in advance, but we carefully observe that $[0]_{n\text{SSS}}$ is used nowhere else and, hence, the simulator can simply overwrite this earlier segment of the transcript that it is generating. In $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$, the same approach is used, simulating that the corresponding preprocessing returns $[v-\hat{v}]_{\mathcal{S}'}$, this time from invoking its generic pseudo-random secret sharing. We note that while in all cases, an incorrect sharing is returned as it does not represent zero, this fact remains hidden from $\mathcal{A}$ as per the underlying LinSSS's privacy property. Now, proceeding to follow the protocol specification, the simulator computes and opens $[v']_s = [\hat{v}]_s + [v-\hat{v}]_s$ so that it holds that $v' = \hat{v} + v - \hat{v} = v$, matching the corresponding expected behavior in the real world.

Higher Domain: The more difficult case is to output from the higher domain. For an expected output v which the simulator learns as part of its input, it again first uses $s.\text{rec}$ for the corresponding LinSSS s to recover the current value $\hat{v}$ according to the simulation. Regarding $\Pi_{\text{MPC}(\text{RSS},\text{3ASS})}$, the single corrupted party $\mathcal{P}_i$ holds share $[\hat{v}]^i_{\text{RSS}} = (\hat{v}_i, \hat{v}_{i-1})$. As shown in [21], it suffices to simply compute $v - \hat{v}_i - \hat{v_{i-1}}$ and let $\mathcal{P}_{i-1}$ send this to the corrupt party instead of $\hat{v}_{i+1}$. This leads to the correct reconstruction of output v. For $\Pi_{\text{MPC}(t\text{SSS},n\text{SSS})}$, we follow [56] by keeping $t-1$ shares, including all shares of the corrupt parties and potentially additional shares of simulated honest parties, and changing the remaining shares.[14] Note that $t-1$ shares together with the anticipated output v define a unique polynomial $f(X)$ of degree $t-1$. We set the share of each party $\mathcal{P}_i$ to $[v]^i_{t\text{SSS}} = f(\alpha_i)$ (cf. §III-D) which keeps the shares of all corrupt parties unchanged. Then, we let the simulator follow the output procedure, which results in output v and maintains privacy as per [56]. For $\Pi_{\text{MPC}(M(\mathcal{S}'),\mathcal{S}')}$, the simulator computes $\hat{\lambda_v} = v - m_v$. It then computes a random sharing $[\hat{\lambda_v}]_{\mathcal{S}'}$ subject to $[\hat{\lambda_v}]^i_{\mathcal{S}'} = [\lambda_v]^i_{\mathcal{S}'}$ for $t-1$ parties $\mathcal{P}_i$, including all corrupt parties, using the linearity of the scheme.[15] The simulator proceeds to follow the output procedure, but now using $[\hat{\lambda_v}]_{\mathcal{S}'}$ instead of $[\lambda_v]_{\mathcal{S}'}$, revealing $\hat{\lambda_v}$, which results in output $m_v + \hat{\lambda_v} = v$.

This simulator is efficient as it emulates the efficient protocol directly and only deviates at the beginning and end of the protocol, again in an efficient fashion. We also note

[14]The choice of the $t-1$ shares to keep is consistent across all outputs.

[15]The choice of the $t-1$ parties is consistent across all outputs.

that for outputs in the higher domain, the deviations to ensure that corrupt parties receive the anticipated outputs are deterministic, given that we fix $t-1$ shares, including those held by the corrupt parties. This ensures consistency across the sent shares for multiple outputs that linearly depend on each other as per the circuit (cf. [56]). ∎

APPENDIX C
OPTIMIZER DETAILS

A. Input-Gadgets and Output-Gadgets for the Trivial Cases

Recall that in §IV, we assumed that $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'} < \mathrm{C}_{\mathrm{in}}^{\mathcal{S}} < \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$ and $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}} < \mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} < \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$ for all input and output gadgets for our optimizer. We provide the gadgets adapted to any of the other cases here.

If $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} \leq \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$, it is always optimal to share in $\mathcal{S}$ and, if necessary, later downgrade for free if a value is needed in $\mathcal{S}'$. Hence, we can simply drop support for inputs in $\mathcal{S}'$ which cannot further decrease communication. We simply omit the input gadget completely, given that there is no need for the input in $\mathcal{S}$ to be followed by an upgrade at some point. Similarly, if $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} \leq \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$, it is always optimal to output from $\mathcal{S}'$, and we omit our output gadgets; if needed, free downgrades can be placed. This happens for $\mathrm{MSS}(A)$ as shown in Tab. I.

Otherwise, if $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} > \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$ but $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}} \geq \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}$, providing an input in $\mathcal{S}'$ and immediately upgrading will never be worse than providing an input directly in $\mathcal{S}$. In this case, we use the input gadget from §IV, but set the weight of its edge to $\mathrm{C}_{\mathsf{up.}}$ so that cutting it equals placement of the required upgrade gate. Regarding outputs, if $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}} < \mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$}$ but $\mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} \geq \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$, re-randomizing and outputting from $\mathcal{S}'$ is not cheaper then to simply upgrade to $\mathcal{S}$ and output from there. Hence, we set the weight of the gadget's edge to $\mathrm{C}_{\mathsf{up.}}$ so that it is cut when an upgrade is required, while all outputs then are in $\mathcal{S}$.

This concludes all possible cases, noting that the case distinction for inputs and outputs is to be made separately.

B. Full Optimizer-Algorithm Specification

Our optimizer introduced in §V consists of three parts: generating a graph G from a circuit $\mathcal{C}$, computing a minimum cut on G, and then deriving the optimal extended circuit $\mathcal{C}'$ from $\mathcal{C}$ and the computed cut. We fully specify the generation of G in Algorithm 1. Here, we encode each directed edge from node g to node g' with weight w as triple (g, g', w). We compute the minimum cut on G using Edmonds-Karp [65] to first compute a maximum flow, and then we convert that into a minimum cut [66]. This partitions the nodes V of G into sets L, R where L contains all nodes that are reachable from the source in the residual network; cut edges are those going from a node in L to a node in R. Finally, deriving $\mathcal{C}'$ from the minimum cut is formally specified in Algorithm 2.

Algorithm 1: Computing graph to be cut.

Data: Circuit $\mathcal{C}$, costs metric COMM with per-gate costs $\mathrm{C}_{\mathrm{in}}^{\mathcal{S}'}, \mathrm{C}_{\mathrm{in}}^{\mathcal{S}}, \mathrm{C}_{\mathsf{up.}}, \mathrm{C}_{\$}, \mathrm{C}_{\mathrm{out}}^{\mathcal{S}'}, \mathrm{C}_{\mathrm{out}}^{\mathcal{S}}$
Result: Directed weighted graph $G = (V, E)$
$G = (V, E) \leftarrow (\{\text{source}, \text{sink}\}, \varnothing)$;
`/* Handle all gates */`
foreach $g \in \mathcal{C}$ *in topological order* **do**
  if g *is input gate* **then**
    $V \leftarrow V \cup \{g^1, g^2\}$; `/* even if no gadget is needed; then, we just omit the edge */`
    if $\mathrm{C}_{in}^{\mathcal{S}'} < \mathrm{C}_{in}^{\mathcal{S}} < \mathrm{C}_{in}^{\mathcal{S}'} + \mathrm{C}_{\mathsf{up.}}$ **then** $E \leftarrow E \cup \{(g^1, g^2, \mathrm{C}_{\mathrm{in}}^{\mathcal{S}} - \mathrm{C}_{\mathrm{in}}^{\mathcal{S}'})\}$;
    else if $\mathrm{C}_{in}^{\mathcal{S}'} + \mathrm{C}_{\mathsf{up.}} < \mathrm{C}_{in}^{\mathcal{S}}$ **then** $E \leftarrow E \cup \{(g^1, g^2, \mathrm{C}_{\mathsf{up.}})\}$;
    $E \leftarrow E \cup \{(\text{source}, g^1, \infty)\}$;
  else if g *is output gate* **then**
    $V \leftarrow V \cup \{g^1, g^2\}$; `/* even if no gadget is needed; then, we just omit the edge */`
    $h_g \leftarrow g^1$;
    if $\mathrm{C}_{out}^{\mathcal{S}} < \mathrm{C}_{out}^{\mathcal{S}'} + \mathrm{C}_{\$} < \mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{out}^{\mathcal{S}}$ **then** $E \leftarrow E \cup \{(g^1, g^2, \mathrm{C}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{C}_{\$} - \mathrm{C}_{\mathrm{out}}^{\mathcal{S}})\}$;
    else if $\mathrm{C}_{\mathsf{up.}} + \mathrm{C}_{out}^{\mathcal{S}} < \mathrm{C}_{out}^{\mathcal{S}'} + \mathrm{C}_{\$}$ **then** $E \leftarrow E \cup \{(g^1, g^2, \mathrm{C}_{\mathsf{up.}})\}$;
    $E \leftarrow E \cup \{(g^2, \text{sink}, \infty)\}$;
  else if g *is multiplication gate* **then**
    $V \leftarrow V \cup \{g_l, g_r\}$;
    $h_g \leftarrow g_r$;
    $E \leftarrow E \cup \{(\text{source}, g_l, \infty), (g_r, \text{sink}, \infty)\}$;
  else
    $V \leftarrow V \cup \{g\}$;
    $h_g \leftarrow g$;
`/* Handle all wires */`
foreach $g \in \mathcal{C}$ *in topological order* **do**
  if g *is input gate* **then**
    foreach *wire* $(g, g') \in \mathcal{C}$ **do** $E \leftarrow E \cup \{(g_2, h_{g'}, \infty)\}$;
  else
    if g *is multiplication gate* **then** $g \leftarrow g_r$;
    if g *has multiple output wires* **then**
      $V \leftarrow V \cup \{g^*\}$;
      $E \leftarrow E \cup \{(g, g^*, \mathrm{C}_{\mathsf{up.}})\}$;
      foreach *wire* $(g, g') \in \mathcal{C}$ **do** $E \leftarrow E \cup \{(g^*, h_{g'}, \infty)\}$;
    else if g *has single output wire* $(g, g') \in \mathcal{C}$ **then**
      $E \leftarrow E \cup \{(g, h_{g'}, \mathrm{C}_{\mathsf{up.}})\}$;
return $G = (V, E)$

C. Proof of Correctness and Optimality

We here provide the full proof for Theorem 3.

Proof: We first show that our optimizer only produces valid sharing assignments. Algorithm 2 uses a minimum cut partition $V = L \dot{\cup} R$ on the output of Algorithm 1 to derive extended circuit $\mathcal{C}'$ from circuit $\mathcal{C}$. First, it is easy to see that if the resulting sharing assignment is correct, the circuit is an extension of $\mathcal{C}$ as it only adds upgrades and downgrades according to the defined rules. For each gate g that is no input or output, Algorithm 2 uses label $\mathcal{S}'$ on output wires of $g \in L$ and label $\mathcal{S}$ on output wires of $g \in R$ (g_l instead of g in case of multiplications). Furthermore, for all gates g' that are reachable from g with a wire, the input wire to g' will have label $\mathcal{S}'$ if $g' \in L$ and label $\mathcal{S}$ if $g' \in R$. If g, g' use different labels, they are in different partitions and an upgrade or downgrade is inserted between them accordingly. From that, we derive that linear operations have inputs and outputs in the same sharing semantics. Furthermore, multiplications g have inputs in $\mathcal{S}$ and outputs in $\mathcal{S}'$ as it must always hold that $g_l \in L, g_r \in R$ as these are connected to the source/sink by edges of infinite weight in G.

Regarding inputs g, we observe that $g^1 \in L$ is connected to the source with infinite weight. For each gate g' connected

Algorithm 2: Computing optimal extended circuit from minimum cut.

Data: Circuit $\mathcal{C}$, costs metric COMM with per-gate costs $\mathrm{c}_{\mathrm{in}}^{\mathcal{S}'}, \mathrm{c}_{\mathrm{in}}^{\mathcal{S}}, \mathrm{c}_{\mathsf{up.}}, \mathrm{c}_{\$}, \mathrm{c}_{\mathrm{out}}^{\mathcal{S}'}, \mathrm{c}_{\mathrm{out}}^{\mathcal{S}}$, directed weighted graph $G = (V, E)$, minimum cut partition $V = L \dot\cup R$
Result: $\mathcal{C}' \in \mathrm{Ext}(\mathcal{C})$ with minimal COMM$(\mathcal{C}')$
foreach $g \in \mathcal{C}$ *in topological order* **do**
  if g *is input gate* **then**
    if $\mathrm{c}_{in}^{\mathcal{S}'} < \mathrm{c}_{in}^{\mathcal{S}} < \mathrm{c}_{in}^{\mathcal{S}'} + \mathrm{c}_{\mathsf{up.}}$ **then**
      if $g^1 \in L \wedge g^2 \in R$ **then** add label $\mathcal{S}$ to all wires starting at g;
      else add label $\mathcal{S}'$ to all wires starting at g;
    else if $\mathrm{c}_{in}^{\mathcal{S}'} + \mathrm{c}_{\mathsf{up.}} < \mathrm{c}_{in}^{\mathcal{S}}$ **then**
      if $g^1 \in L \wedge g^2 \in R$ **then**
        replace g by upgrade-gate;
        add label $\mathcal{S}$ to all wires starting at new upgrade-gate;
        add copy of prior g and $\mathcal{S}'$ wire from g to upgrade;
      else add label $\mathcal{S}'$ to all wires starting at g;
    else add label $\mathcal{S}$ to all wires starting at g;
    if all output wires are labeled $\mathcal{S}$ and there exist wires from g to non-multiplication gate $g' \in L$, add downgrade-gate, add $\mathcal{S}$ edge from g to downgrade, and rewire all wires from g to $g' \in L$ to $\mathcal{S}'$ wires from downgrade to g';
  else if g *is output gate* **then**
    if $\mathrm{c}_{\mathsf{up.}} + \mathrm{c}_{out}^{\mathcal{S}} < \mathrm{c}_{out}^{\mathcal{S}'} + \mathrm{c}_{\$} \wedge g^1 \in L \wedge g^2 \in R$ **then**
      replace g by upgrade-gate;
      add copy of prior g and $\mathcal{S}'$ wire from upgrade to g;
    `/* outputs from S handled implicitly by next block */`
  else
    if g *is multiplication gate* **then** $g' \leftarrow g_l$;
    else $g' \leftarrow g$;
    if g *has multiple output wires* **then**
      if $g' \in L \wedge g^* \in R$ **then**
        add upgrade-gate and $\mathcal{S}'$ wire from g to upgrade;
        foreach $(g^*, g'', \cdot) \in E$ **do**
          if $g'' \in R$ **then**
            replace wire (g, g'') by $\mathcal{S}$ wire from upgrade to g'';
          else add label $\mathcal{S}'$ to wire (g, g'');
      else if $g' \in R \wedge g^* \in L$ **then**
        add downgrade-gate and $\mathcal{S}$ wire from g to the downgrade;
        foreach $(g^*, g'', \cdot) \in E$ **do**
          replace wire (g, g'') by $\mathcal{S}'$ wire from downgrade to g'';
      else if $g' \in L \wedge g^* \in L$ **then**
        add label $\mathcal{S}'$ to all wires (g, g'');
      else
        foreach $(g^*, g'', \cdot) \in E$ **do**
          if $g'' \in L$ **then**
            replace wire (g, g'') by $\mathcal{S}$ wire from g to new downgrade, add $\mathcal{S}'$ wire from downgrade to g'';
          else add label $\mathcal{S}$ to wire (g, g'');
    else
      let (g, g'') be the single output wire of g and $(g', g^*, \cdot)$ be the corresponding edge in E;
      if $g' \in L \wedge g^* \in R$ **then**
        replace wire (g, g'') by $\mathcal{S}'$ wire from g to new upgrade and $\mathcal{S}$ wire from upgrade to g'';
      else if $g' \in R \wedge g^* \in L$ **then**
        replace wire (g, g'') by $\mathcal{S}$ wire from g to new downgrade and $\mathcal{S}'$ wire from downgrade to g'';
      else if $g' \in L \wedge g^* \in L$ **then** add label $\mathcal{S}'$ to wire (g, g'');
      else add label $\mathcal{S}$ to wire (g, g'');
return $\mathcal{C}'$ *(result of previous modifications to* $\mathcal{C}$*)*

with g: if $g' \in R$, the input to g' will be in $\mathcal{S}$ by inputting in $\mathcal{S}$ or adding an upgrade. The same holds for multiplication gates as $g'_r \in R$ always. Furthermore, if $g' \in L$ while the input is in $\mathcal{S}$ or upgraded, a downgrade is added accordingly for g'. Outputs inherit the semantics of their input wires except for if they are fed by a wire in $\mathcal{S}'$ and upgrading and an output from $\mathcal{S}$ is cheaper than re-randomizing and outputting from $\mathcal{S}'$, in which case an upgrade is added and appropriate labels are selected.

Next, we investigate optimality. Each edge, apart from edges in input and output gadgets, has infinite weight or can be cut while having weight $\mathrm{c}_{\mathsf{up.}}$. Exactly each of these edges being cut yields one upgrade in our extended circuit, costing communication $\mathrm{c}_{\mathsf{up.}}$. If inputs in $\mathcal{S}$ are not more expensive than inputs in $\mathcal{S}'$, it is optimal to always input in $\mathcal{S}$ as downgrades are free and, hence, versions of the input in any domain can be obtained at no additional cost. If outputs from $\mathcal{S}'$ (including re-randomization) are not more expensive than outputs from $\mathcal{S}$, we can also always use that and, if necessary, add a free downgrade. In both cases, the corresponding gadget does not have an edge that can be cut as no decision is to be made. Otherwise, having an input eventually in $\mathcal{S}$ is more expensive, costing more $\mathrm{c}_{\mathrm{in}}^{\mathcal{S}} - \mathrm{c}_{\mathrm{in}}^{\mathcal{S}'}$ for inputting in $\mathcal{S}$ or $\mathrm{c}_{\mathsf{up.}}$ to immediately upgrade, whatever costs less. Likewise, having an output being fed in $\mathcal{S}'$ is more expensive, costing more $\mathrm{c}_{\mathrm{out}}^{\mathcal{S}'} + \mathrm{c}_{\$} - \mathrm{c}_{\mathrm{out}}^{\mathcal{S}}$ for re-randomizing and outputting from $\mathcal{S}'$ instead of $\mathcal{S}$, or $\mathrm{c}_{\mathsf{up.}}$ for having to upgrade to $\mathcal{S}$ first, whatever costs less. The respective additional costs are set as edge weights.

Note that batching the upgrade of multiple of a gate's output wires as allowed in §III-B is implicitly covered by using intermediate nodes g^* in Algorithm 2 after each gate with multiple outputs: any assignment, that has multiple separate upgrade gates for the same result of a gate, must be worse than any of the assignments created by the optimizer, as the optimizer never introduces multiple upgrades for one gate result.

Now, assume that our optimizer output is not optimal for a circuit $\mathcal{C}$, i.e., that there exists an extended circuit $\mathcal{C}^*$ better than $\mathcal{C}'$. By the choice of input/output domains and placement of upgrades, we can derive a set of cut edges in G according to $\mathcal{C}^*$. We know that the set of cut edges associated to $\mathcal{C}'$ in the same way must be the minimal capacity set of cut edges by the correctness of maximum flow computation and because of the max-flow min-cut theorem [66]. Therefore, the set of cut edges associated to $\mathcal{C}^*$, if it is smaller, cannot be a cut of the graph. Hence, there exists a path from source to sink in the residual network of $\mathcal{C}^*$. If this is the path from g_l of one multiplication to g'_r of another multiplication (extended to source and sink), there exists a path from g over linear gates to g' with no upgrade. The output of g must be in $\mathcal{S}'$, as must be the output of all linear gates in-between, as their inputs and outputs must be consistent. Then, g' receives its inputs in $\mathcal{S}'$, which contradicts $\mathcal{C}^*$ being a valid extended circuit. If there is a path over an input gadget to g_r of a multiplication, the input is in $\mathcal{S}'$, followed by a path of linear operations, remaining in $\mathcal{S}'$, finally arriving at g, which is invalid. Symmetrically, a path

from g_l of a multiplication over an output gadget corresponds to $\mathcal{S}'$ being used from g following linear operations up to the output. Yet, the output is in $\mathcal{S}$, yielding a contradiction. For a path over an input and an output gadget, the input would be in $\mathcal{S}'$, and the output would be in $\mathcal{S}$ while again, linear operations in-between cannot change the sharing semantics. Hence, in any case, a smaller set of cut edges implies that the underlying share assignment has an inconsistency at some point. ■

D. Future Directions for Performance Improvements

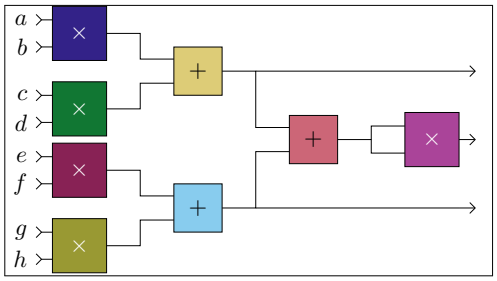
The complexity of our optimizer algorithm may be improved both asymptotically and concretely. The most expensive step of our algorithm is the flow computation on the modified flattened circuit graph. The circuit may be understood as a hypergraph: if several wires are emitted from one gate in our algorithm, we may instead represent them as one directed hyperedge in a weighted hypergraph. Reducing the hypergraph to a simple graph using Lawler's transformation [63], for computing the maximum flow, corresponds to the behavior of our algorithm in §C-B. Instead, we may use maximum flow algorithms specifically designed for hypergraphs [74], yielding better asymptotic complexity. It is also possible to partition the graph of the flow problem into several connected components and compute the flow in parallel for each of them. For circuits where computations do not skip between more than two multiplication layers, this partitioning of nodes may be significant. This improvement, however, requires the overhead of partitioning the graph. It is therefore only useful for large and deep graphs with many layers of multiplications, such that no linear operation result is used later than the next layer.

Our implementation opts instead for a simple flow computation as described in §IV, as we remain in a practical range of run times for our circuits and therefore other algorithms would require more complicated implementations than necessary for our purposes.

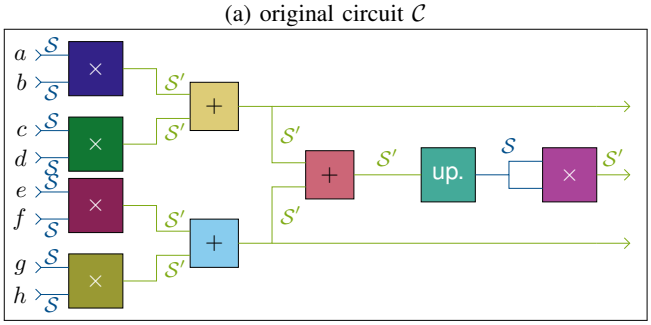
E. Recursively Optimizing Preprocessing for Masked LLSS

We provide additional information on the preprocessing optimization when working with a masked LLSS as outlined in §III-E2 here. While we were not able to optimize the preprocessing of ABY2.0 [28] but only the online phase, we now see that our masking compiler for LLSS can also improve preprocessing. Example settings where this is possible are 2PC with a dealer (ASTRA [27]), and 3PC with $\mathcal{S}' = \text{RSS}$ (SWIFT [29]). For a concrete evaluation of the preprocessing optimization, see §D-A.

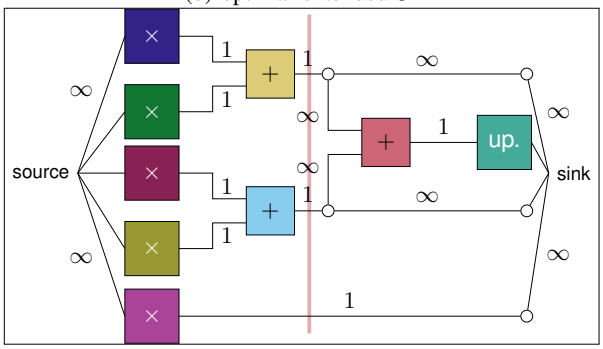
Given a circuit $\mathcal{C}$ (e.g., Fig. 11a), we first run our original optimization (cf. §IV) to optimize online communication, resulting in an extended circuit (e.g., Fig. 11b). For a multiplication $[\![x]\!] \cdot [\![y]\!]$, we do not immediately add the term $\langle \lambda_x \lambda_y \rangle$ to the result, yielding an error in the value represented by the resulting $\mathcal{S}'$-sharing (e.g., Fig. 11d in red). Errors then accumulate while computation in $\mathcal{S}'$ continues. We insert explicit "fix" gates which compensate the error at some point later while still in $\mathcal{S}'$ so that no errors reach an output (from $\mathcal{S}'$) or upgrade gate. Each of these gates simply subtracts the accumulated error non-interactively, while we can compute the required correction during preprocessing, noting that it is a linear combination of masks and products of masks.



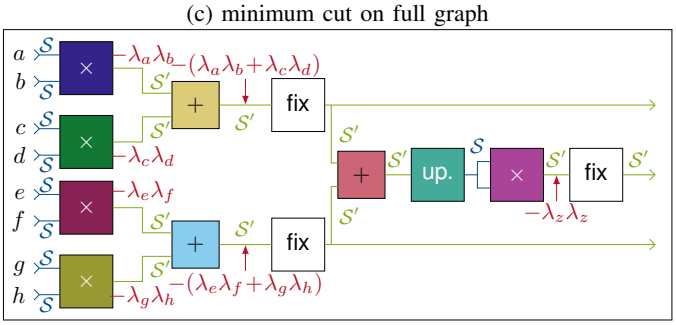
(a) original circuit $\mathcal{C}$



(b) optimal extended $\mathcal{C}'$



(c) minimum cut on full graph



(d) extended circuit $\mathcal{C}'$ plus propagating errors (in red) and error correction fixes

Figure 11: Example preprocessing optimization of extended circuit $\mathcal{C}'$. Same coloring used for individual gates.

If we would simply evaluate the circuit as first introduced in §IV, this would be the equivalent of always fixing immediately after multiplications, computing $\mathcal{S}'$-sharings of the prod-

ucts of two $\mathcal{S}'$-sharings of masks during preprocessing. Yet, if $\mathcal{S}'$ itself is the higher domain of an LLSS $(\mathcal{S}', \mathcal{S}'')$, then each fixing has the cost of only one upgrade in this LLSS: The errors resulting from each multiplication can be non-interactively computed in $\mathcal{S}''$, further arbitrary linear computation is done in $\mathcal{S}''$, and where fixing is intended, the accumulated error (correction) is upgraded to $\mathcal{S}'$ to later be used by our main LLSS. Similarly, if a dealer is present[16] (cf. Remark 3.1, p. 8), it has access to all masks and can hence locally compute and then $\mathcal{S}'$-share the error corrections for each fixing.

Using a similar approach as in §IV, we now aim to minimize the number of required fixing operations. We must ensure that on the (linear computation) path between each multiplication output and upgrade or output from $\mathcal{S}'$, there is a fixing operation. We model this by a directed graph (e.g., Fig. 11c) where we connect all multiplications to a source, remove their other input wires, and connect all upgrade gates and $\mathcal{S}'$-outputs to a sink while removing output wires of the upgrades. For gates with multiple output wires, we use the same gadgets as in §IV with infinite weights for all but the first edge. We assign infinite weight to all edges from/to source and sink and use uniform weight 1 for all remaining edges, given that the only operation we aim to insert is the fixing, which has constant cost. Then, we compute a minimum cut. As this contains the minimum number of edges separating multiplication outputs from upgrades and $\mathcal{S}'$-outputs, the cut edges are the optimal position to insert fixing gates (e.g., Fig. 11d).

We remark that for sub-circuits which solely receive their inputs from multiplications and provide their outputs to upgrades, i.e., that are not connected over linear gates to any input or output, fixing always immediately before the upgrades is optimal. This is the case as a smaller cut at an earlier location would also offer a better option to place the upgrades, contradicting the online optimality of the used extended circuit. Yet, optimizing online while also considering inputs and outputs for further optimization can yield other cases, like in Fig. 11, making our more complex preprocessing optimizer necessary. Still, for sub-circuits completely between layers of multiplications and upgrades, the optimization can be improved using the aforementioned observation.

APPENDIX D
ADDITIONAL EVALUATION RESULTS

A. Preprocessing Cost and Evaluation of Preprocessing Optimization from §C-E

While our LLSS-protocol from §III-C based on replicated secret sharing (RSS) runs the entire computation online, the protocols from §III-D based on Shamir's secret sharing [17] (tSSS) and §III-E ($M(\mathcal{S}')$ for LinSSS $\mathcal{S}'$) based on masked secret sharing utilize preprocessing. In this section, we provide a supplementary evaluation of preprocessing performance of the affected protocols.

Regarding the protocol from §III-D based on tSSS, we recall that preprocessing is necessary to compute a triple $(\llbracket r \rrbracket, \langle r \rangle, \mathcal{P}_i)$ for each upgrade. Optimizing the online phase aims to decrease the number of upgrades (besides optimizing inputs and outputs), so that it automatically also improves the preprocessing. We provide the resulting communication for $n = 10$ parties in Tab. V. For most circuits, improvement regarding online and preprocessing is similar. Differences can be explained by online communication also depending on inputs and outputs to the circuit which are included in our optimization. For MSE, we observe a reduction of preprocessing cost to zero, because the circuit does not contain paths with multiple multiplications, so that no upgrades are needed for our optimization. For Neural-Net, preprocessing is not improved, as the online improvements stem from optimizing inputs and outputs rather than upgrades for this circuit. Generally, we observe that the majority of communication cost stems from the online phase, whereas preprocessing has a significantly lower performance impact. We recall from Tab. I on p. 10 that the online cost per upgrade is $\approx 1.5n$ field elements and further cost is introduced by inputs and outputs, whereas the amortized preprocessing cost per upgrade only is $\approx 0.5n$. Interestingly, for $n = 3$, preprocessing even is fully non-interactive with the PRF optimizations from §A-B2.

Table V: Online and preprocessing communication in bits for $n = 10$-party computation based on tSSS (§III-D) on various circuits, using as baseline the protocol from [16] that immediately upgrades after each multiplication, and our LLSS-protocol extension.

Circuit	online			preprocessing		
	base. ([16])	**ours** (§III-D)	**impr.**	base. ([16])	**ours** (§III-D)	**impr.**
RCA	3276	3208	2.07%	372	360	3.22%
PPA	9932	6172	37.85%	1908	948	50.31%
Multiplier	59 800	52 952	11.45%	13 416	11 736	12.52%
float-add.	8852	6744	23.81%	1836	1320	28.10%
float-mult.	44 108	38 812	12.00%	9972	8724	12.51%
float-div.	4 281 316	3 200 116	25.25%	987 228	737 484	25.29%
AES-128	339 456	229 184	32.48%	76 800	50 880	33.75%
SHA-256	1 191 204	1 162 372	2.42%	270 876	264 204	2.46%
LowMC-256	102 224	96 932	5.17%	21 168	19 404	8.33%
MSE	836 160	197 440	76.38%	147 456	0	100%
Neural-Net	1 268 864	1 070 720	15.61%	245 760	245 760	0.00%

For masked secret sharing $M(\mathcal{S}')$ for LinSSS $\mathcal{S}'$ as described in §III-E, we again consider $\mathcal{S}' = 2$ASS and two party computation (2PC), corresponding to ABY2.0 [28] or, if an additional dealer is available, ASTRA [27]. What differs is the instantiation of Π_{multPre} for the preprocessing phase. Without a dealer, we follow [28], using oblivious transfer (OT) to instantiate Π_{multPre}. We optimize this using silent OT [75], following more recent followup works of ABY2.0 [28] such as [55]. Computing batches of 10^7 random OTs with 128-bit security as done in [55] yields amortized preprocessing communication of ≈ 4.236 bits per mult in the binary domain, again following the approach of [55]. In the arithmetic domain, mult can be realized with communication of $\approx 2|\mathcal{R}|(1+|\mathcal{R}|+0.118)$ bits following the improved approach proposed in [76].

We provide the results in Tab. VI. As preprocessing is done per mult and not per upgrade, our optimization from §IV does

[16] even if there is no LLSS $(\mathcal{S}', \mathcal{S}'')$

Table VI: Online and preprocessing communication in bits for two-party computation (with and without a dealer) based on masked sharings $M(\text{2ASS})$ (§III-E) on various circuits, using as baseline the protocols ABY2.0 [28] and ASTRA [27] that immediately upgrade after multiplication, and our LLSS-protocol extensions.

Circuit	**online**			**preprocessing**			
	2PC/ABY2.0 [28] and 2PC+dealer/ASTRA [27]			2PC	2PC+dealer		
	baseline ([28], [27])	**ours** (§III-E)	**impr.**	base. ([28])/**ours**	base. ([27])/**ours**	**ours+opt. from §C-E**	**impr.**
RCA	158	154	2.53%	131	31	31	0.00%
PPA	414	252	39.13%	674	159	106	33.33%
Multiplier	2332	2052	12.00%	4736	1118	1005	10.10%
float-add.	354	268	24.29%	648	153	118	22.87%
float-mult.	1710	1500	12.28%	3520	831	734	11.67%
float-div.	164 730	123 106	25.26%	348 491	82 269	61 521	25.22%
AES-128	13 184	8864	32.76%	27 110	6400	4368	31.75%
SHA-256	46 170	45 055	2.45%	95 619	22 573	22 025	2.42%
LowMC-256	4120	3717	9.78%	7472	1764	1764	0.00%
MSE	147 584	49 280	66.60%	6 401 360	49 152	64	99.87%
Neural-Net	214 656	164 480	23.37%	10 668 933	81 920	74 368	9.21%

not improve preprocessing. The baseline and our modified protocol share the exact same preprocessing cost. Without a dealer, the reliance on instantiating Π_{multPre} with OT yields a relatively expensive preprocessing phase with approximately double the communication cost of the online phase for binary circuits. The RCA is an exception, as the majority of its online costs stems from inputs and outputs. For the arithmetic domain, preprocessing becomes significantly more expensive. If we use a dealer as in ASTRA [27], preprocessing communication is decreased to one bit or ring element per mult, yielding approximately half of the online communication for binary and arithmetic circuits. Furthermore, the dealer now enables us to use the optimization first sketched in §III-E2 and then fully introduced in §C-E. As previously for tSSS, preprocessing improvement is similar to online improvement in most cases. The structure of the RCA and LowMC-256 do not enable any aggregation of "fixing"-operations for the preprocessing optimization. On the other hand, for MSE, 768 mult are computed and the results summed up, enabling to aggregate to a single "fix" on the final sum to improve by a factor of $768\times$.

B. LLSS-MPC Based on Shamir's Secret Sharing for Different Numbers of Parties

In Tab. VII we compare the improvement in online communication calculated when optimizing the multi-party protocols with Shamir's Secret Sharing [17] outlined in §III-D for different numbers of parties $n = 3$ and $n = 10$. As expected, our optimization yields very similar results in both settings. Yet, as visible in Tab. I, for higher n the upgrade cost grows faster than the input/output costs, causing the optimizer to prefer inputs in higher domain $\mathcal{S}$ to save on upgrades more often. While not considered in this optimization, it is also noteworthy that for $n = 3$, the preprocessing becomes non-interactive except for the static exchange of pre-shared keys, using the PRF optimizations from §A.

C. Improvement Without Inputs and Outputs in Lower Domain

One may question how much of our improvements stem from reducing the number of upgrades within the circuit alone and how much stems from additionally enabling to freely use inputs and outputs in domains $\mathcal{S}$ or $\mathcal{S}'$. To investigate this, we force all inputs and outputs to be in the higher domain $\mathcal{S}$, which can be achieved by removing all input gadgets and setting the edge weights of output gadgets to $\mathsf{C}_{\mathsf{up}.}$ in our optimizer construction from §IV. Now, all finite edge weights are uniform, so that we simply count the number of required upgrades, given that the cost for these upgrades is simply obtained by multiplying this number by the individual upgrading cost $\mathsf{C}_{\mathsf{up}.}$. We provide the number of required upgrades for our optimization results in Tab. VIII.

The results in most cases are comparable to those in Tab. III, which also include and optimize for inputs and outputs. For RCA and LowMC-256, we observe that there is no improvement, i.e., all prior measured improvements only stem from also utilizing input and output layers. E.g., for RCA, this is the case as the circuit is a chain of full adders, each containing a single AND-gate, where all ANDs are aligned on the circuit's critical path. Regarding MSE, we observe the improvement in upgrade gates to be significantly higher than the overall communication improvement from Tab. III. This is because all 1280 multiplications can be merged into a single

Table VII: Total online communication in bits for various circuits using the baseline BGW-style protocol [16] that immediately upgrades after multiplication, and our LLSS-protocol extensions from §III-D, using different numbers of parties n. Secret-sharing threshold t set accordingly to $t = \lfloor (n+1)/2 \rfloor$ for honest majority setting.

Circuit	$n = 3, t = 2$			$n = 10, t = 5$		
	base. ([16])	**ours** (§III-D)	**impr.**	base. ([16])	**ours** (§III-D)	**impr.**
RCA	378	370	2.11%	3276	3208	2.07%
PPA	1146	716	37.52%	9932	6172	37.85%
Multiplier	6900	6114	11.39%	59 800	52 952	11.45%
float-add.	1014	772	23.86%	8852	6744	23.81%
float-mult.	5082	4472	12.00%	44 108	38 812	12.00%
float-div.	493 998	369 254	25.25%	4 281 316	3 200 116	25.25%
AES-128	39 168	26 464	32.43%	339 456	229 184	32.48%
SHA-256	137 486	134 160	2.42%	1 191 204	1 162 372	2.42%
LowMC-256	11 768	11 180	4.99%	102 224	96 932	5.17%
MSE	196 736	49 344	74.92%	836 160	197 440	76.38%
Neural-Net	296 576	247 040	16.70%	1 268 864	1 070 720	15.61%

Table VIII: Number of inserted upgrade operations in the baseline sharing vs with bottleneck share assignment while all input and output operations are fixed to higher sharing.

Circuit	baseline	**ours** (§III)	**impr.**
RCA	31	31	0.00%
PPA	159	106	33.33%
Multiplier	1118	1005	10.10%
float-add.	153	118	22.87%
float-mult.	831	734	11.67%
float-div.	82 269	61 521	25.22%
AES-128	6400	4368	31.75%
SHA-256	22 573	22 025	2.42%
LowMC-256	1764	1764	0.00%
MSE	768	1	99.87%
Neural-Net	1280	1162	9.21%

dot product, whereas especially many inputs still require communication where less optimization is possible. For Neural-Net, we get a smaller improvement here than in Tab. III, which we attribute to the large input layer, providing many additional opportunities for optimization if any domain can be used for inputs. We conclude that our additional flexibility regarding inputs and outputs open good opportunities for optimization, whereas even without this flexibility, good improvement is possible in most cases. This is also important should a circuit be used as a component of a larger circuit where inputs and outputs are not secret-shared respectively opened, but instead already are sharings provided by and handed back to the larger circuit. In this case, we could always require inputs and outputs to be in $\mathcal{S}$, ensuring that the optimized sub-circuit can seamlessly be composed with other sub-circuits. Yet, this *hard boundary* adds additional restrictions and more expensive optimization over the larger circuit may yield better results.

We also observe that the improvements in Tab. VIII match those in Tab. VI from §D-A for optimizing the preprocessing of MPC based on masked secret sharing. Here, upgrades need to split each path from a multiplication to another multiplication or an output from the lower domain, whereas the preprocessing optimization from §C-E needs to split each path from a multiplication to an upgrade or an output from the lower domain by a fixing operation. The only difference is splitting before multiplications instead of already placed upgrades. As upgrades are assumed to be placed optimally, the restriction of choices in the second case appears not to decrease the quality of the optimization output. Yet, we recall that when also considering inputs and outputs from any domains, the placement of upgrades can differ from that in this section. Hence, we hypothesize that the observed similarity is to be expected, but that there may be other examples where no strict equality is given.

D. Comparison to MAESTRO [44] and ALKAID [45]

The recent works MAESTRO [44] and ALKAID [45] both utilize a second secret-sharing domain beyond directly upgrading back to the main domain. MAESTRO decouples non-interactive multiplications from upgrades on replicated and additive sharings as we do for $(\mathrm{RSS}, \mathrm{3ASS})$ in §III-C. ALKAID places XOR gates after AND gates and only upgrades the result of the XOR instead of that of multiple ANDs. Furthermore, it uses three levels: masked shares based on replicated sharings $M(\mathrm{RSS})$, replicated sharings RSS, and additive sharings 3ASS, yielding a *Multi-LLSS* (§III-F).

In contrast to our work, these works utilize such optimizations to hand-optimize for specific applications. MAESTRO [44] optimizes AES in MPC, also relying on further optimizations such as lookup tables (LUTs) based on one-hot vectors. ALKAID [45] hand optimizes parallel prefix adders (PPAs) using its three levels, and based on these PPAs, conversions and activation functions. In contrast, our work is generic. We formalize and generalize the notion of Leveled Linear Secret Sharing (§III-A), and we provide an optimizer (§IV) that optimally and automatically improves any input circuit, also utilizing inputs and outputs in any domain.

Given that [44], [45] use additional building blocks for their hand optimization, i.e., LUTs respectively more than two levels (which is currently not supported by our optimizer), it is expected that these protocols, being results of long lines of work to optimize highly specialized building blocks, outperform our generic solution. Yet, our approach provides a simple, automated solution without requiring hand-optimizing circuits. In the following, we empirically compare communication complexities between the different approaches:

1) Comparison with MAESTRO [44] for AES: Tab. IX compares the communication complexities between naïve evaluation of a Boolean AES circuit [68] using replicated secret sharing [21], our optimized version of that circuit using LLSS, the predecessor of MAESTRO by Chida et al. [77], and different variants of MAESTRO [44]. We assume here that inputs are already provided in secret-shared format (higher domain where applicable) and do no final reconstruction, benchmarking circuit evaluation only without input and output phases. All protocols use three non-colluding parties and we consider their semi-honest variants here. They all rely on variants and extensions of MPC on replicated secret sharing [21]. As the circuit from [68] includes the key expansion, we also include this for all other benchmarks. The variants of MAESTRO are as described in [44, Section 4]. For variant "$(3,3)$ LUT-256", [44] does not explicitly define how to run the key expansion. Hence, we include two variants, "r" which is optimized for rounds and uses "$(2,3)$ LUT-256" for the S-boxes during key expansion, and "c" which is optimized for total communication and uses the "$GF(2^4)$" approach for the key expansion S-boxes.

As expected, the special-purpose protocol MAESTRO [44] offers the best solutions for low online or total communication and low rounds, also providing more fine-grained tradeoffs with its different variants. Yet, interestingly, our optimized LLSS version has the second lowest total communication (only 9% higher than the lowest, whereas the RSS baseline is 60% higher). We also outperform [77] which shares the communication with our RSS baseline. Yet, our round complexity is at least double of that of all other approaches, given that LLSS cannot optimize rounds beyond removing a

Table IX: Communication in bits over all parties and rounds for evaluating one block of AES128 encryption, including the key expansion. Inputs and outputs are secret shares.

Protocol	Communication online	Communication total	online rounds
Boolean RSS [21] (circuit from [68])	19 200	19 200	60
Our optimized leveled RSS (§III-C)	13 104	13 104	60
Chida et al. [77]	19 200	19 200	30
MAESTRO [44] LUT-16	9600	16 200	20
MAESTRO [44] $GF(2^4)$	12 000	12 000	30
MAESTRO [44] $(2,3)$ LUT-256	4800	153 000	10
MAESTRO [44] $(3,3)$ LUT-256 (r)	8640	48 840	10
MAESTRO [44] $(3,3)$ LUT-256 (c)	10 080	20 640	30

single round when outputs are allowed in the lower domain, which we do not allow here for fairness. We conclude that our generic approach comes relatively close to the hand-optimized state of the art in the metric it aims to optimize: total communication. In contrast to MAESTRO [44], our approach achieves this optimization automatically in a generic way. Our approach does not enable to move operations to preprocessing, utilize LUTs, etc., so combining such optimizations from MAESTRO [44] and our techniques may enable even better solutions in the future.

2) Comparison with ALKAID [45] for PPAs: Regarding ALKAID [45], we compare performance for 32-bit addition using parallel prefix adders (PPAs) which is what [45] focuses on for optimization. As for MAESTRO [44] above, we do not include input and output phases and assume inputs and outputs to already be provided in secret-shared format (in the higher domain where applicable). All protocols here use three non-colluding semi-honest parties. Results are provided in Tab. X.

Table X: Communication in bits over all parties and rounds for evaluating one 32-bit addition using a parallel prefix adder (PPA). Inputs and outputs are secret shares.

Protocol	Communication online	Communication total	online rounds
Boolean RSS [21] (circuit from [32])	477	477	5
Our optimized leveled RSS (§III-C)	318	318	5
ALKAID [45]	630	935	4

Surprisingly, we do not only find that our communication outperforms ALKAID [45] by $1.98\times$ online and $2.94\times$ total, but that even the naïve baseline of evaluating a PPA circuit in RSS [21] is better than ALKAID. ALKAID's only advantage is a decrease of round complexity from 5 to 4 rounds. We note that these observations contradict [45], which claims also an improvement of communication over the baseline. We discuss that issue in the following.

In Tab. X, we surprisingly observe that ALKAID [45] even has worse communication compared to simply evaluating a PPA circuit in RSS [21] without our optimization. The numbers we report for ALKAID are directly calculated from [45, Lemma 2] for $\ell = 32$. The full proof of this lemma is claimed to be in [45, Appendix B], but that appendix neither exists in the conference version nor in the ePrint version[17], so we could not verify.

ALKAID includes concrete PPA benchmarks in [45, Table IV] for $\ell = 64$-bit addition. They claim to have a communication of 0.141 KB online and 0.198 KB total per PPA, whereas for ABY3 [1], the communication (all online) is claimed to be 0.203 KB. Yet, ABY3 [1] uses the RSS protocol from [21] for semi-honest PPA evaluation. Using [45, Lemma 2], we get that ALKAID should have 0.193 KB online and 0.281 KB total communication, i.e., 37–42% higher values. Meanwhile, the PPA from [32] uses 383 AND gates, so evaluating that with ABY3 [1] would cost $383{\cdot}3$ bits, i.e., 0.144 KB, which is 29% lower than what [45] claims. To conclude, ALKAID [45] appears to report too high communication for [1] and, assuming correctness of [45, Lemma 2], too low communication for their own protocol when observing that they outperform ABY3 [1], whereas they appear to be worse than ABY3.

Appendix E
Future Directions

One possible future research direction is to generalize MPC on LLSS to more than two layers. Possible directions are addressed in §III-F and could allow to decrease round complexity by, e.g., non-interactively multiplying from a higher to an intermediate sharing, then non-interactively multiplying yet again to a lower sharing, and finally upgrading. Yet, this requires proper formalization, security analysis, and an adapted optimizer (cf. §IV) which now needs to solve a more complex problem by assigning one out of more than two sharing semantics to each wire.

As outlined in §I-B, there exist parallels between LLSS and leveled homomorphic encryption [71]. It may be of interest to try rephrasing this or, e.g., constructions based on degree-d homomorphic secret sharing [78], in our framework, generalizing it beyond classical secret-sharing-based MPC. This would also enable utilizing a variant of our optimizer from §IV in these fields.

Finally, while we focus on passive security in this work, similar constructions appear to be possible in the active security model as well. For instance, MPC on replicated sharings [21] can be lifted to active security [23] using zero-knowledge fully linear interactive oracle proofs (zk-FLIOPs) [72]. It is utilized here that the correctness of messages after a multiplication requires to prove a degree-two relation. Correctness when doing additional linear operations on one or more multiplication outputs before interactively upgrading still corresponds to a degree-two relation, now for the messages sent during upgrading. One could combine LLSS with zk-FLIOPs as [72] provides efficient constructions to prove *any* degree-two relation. Yet, this would require proofs for different degree-two relations, potentially making it harder to leverage the desired amortization of zk-FLIOPs. Still, e.g., [73] demonstrates amortization with different degree-two relations by supporting dot products of different dimensions in the same zk-FLIOP. For BGW-style protocols, verifiable secret sharing

[17] https://ia.cr/2025/2298

enables lifting to active security [13], [15], [16] while requiring the use of a lower corruption threshold, which appears to also generalize to our construction. Finally, SWIFT [29] serves as one example to use masked secret sharing in an active security setting. It reaches active security by using replicated sharings as lower sharings, providing redundancy when reconstructing to upgrade back to masked sharings. This mechanism is not restricted to the case where upgrading happens immediately after a multiplication as in the original protocol.