Masked Triples
Amortizing Multiplication Triples across Conditionals

David Heath, Vladimir Kolesnikov, and Stanislav Peceny

Georgia Institute of Technology, Atlanta, GA, USA
{heath.davidanthony,kolesnikov,stan.peceny}@gatech.edu

Abstract. A classic approach to MPC uses preprocessed *multiplication triples* to evaluate arbitrary Boolean circuits. If the target circuit features conditional branching, e.g. as the result of a `IF` program statement, then triples are wasted: one triple is consumed per `AND` gate, even if the output of the gate is entirely discarded by the circuit's conditional behavior.

In this work, we show that multiplication triples can be re-used across conditional branches. For a circuit with b branches, each having n `AND` gates, we need only a total of n triples, rather than the typically required $b \cdot n$. Because preprocessing triples is often the most expensive step in protocols that use them, this significantly improves performance.

Prior work similarly amortized *oblivious transfers* across branches in the classic GMW protocol (Heath et al., Asiacrypt 2020, [HKP20]). In addition to demonstrating conditional improvements are possible for a different class of protocols, we also concretely improve over [HKP20]: their maximum improvement is bounded by the *topology* of the circuit. Our protocol yields improvement independent of topology: we need triples proportional to the size of the program's longest execution path, regardless of the structure of the program branches.

We implemented our approach in `C++`. Our experiments show that we significantly improve over a "naïve" protocol and over prior work: for a circuit with 16 branches and in terms of total communication, we improved over naïve by 12× and over [HKP20] by an average of 2.6×.

Our protocol is secure against the semi-honest corruption of $p-1$ parties.

Keywords: MPC, conditional branching, Beaver triples.

1 Introduction

Secure Multiparty Computation (MPC) enables untrusting parties to compute a function of their private inputs while revealing only the function output. In this work, we consider semi-honest MPC protocols that use the classic trick of Beaver to evaluate Boolean circuits by preprocessing 'multiplication triples' [Bea92].

In such protocols, `XOR` gates are 'free' (i.e., require no interaction), but each `AND` gate consumes a distinct multiplication triple. To generate these triples, the parties use a more expensive MPC protocol that can be run ahead of time in a preprocessing phase. The communication in this phase is proportional to the number of triples and is usually the performance bottleneck. Hence, if we reduce the number of required triples, then we significantly improve performance.

Because one triple is needed per AND gate, protocols waste significant work if the computed function has conditional behavior, e.g. as the result of an IF program statement. Each gate requires a distinct triple, even if the output of the gate is entirely discarded by the function's conditional behavior.

Our protocol re-uses multiplication triples across conditional branches. A single triple can support any number of AND gates, so long as the gates occur in mutually exclusive program branches. This re-use does require that the parties hold additional correlated randomness, but the parties can generate this randomness efficiently. Our approach greatly decreases total communication and hence improves performance.

1.1 High Level Intuition

Multiplication triples typically cannot be re-used (Section 2.5 reviews multiplication triples in detail): triples are essentially one-time-pads on cleartext values in the circuit: since no strict subset of parties knows the values in the triple, it is secure to use the triple to mask cleartext values. However, if we use the same triple for two different gates, then we violate security. We can work around this.

Consider two conditionally composed circuits C^0 and C^1, both with n AND gates. For sake of example, suppose C^0 is the active branch, but suppose the parties do not know and should not learn this fact. We re-use the same triples to evaluate gates in both C^0 and C^1 by carefully applying secret shared *masks* to the triples. For the inactive branch C^1, the parties mask the shares with XOR shares of *uniform* masks, randomizing the triples and preventing us from breaking the security of one-time-pad. By randomizing the triples, we violate the correctness of AND gates on the inactive branch, but this is of no concern: the output of each inactive AND gate is ultimately discarded. For the active branch C^0, the parties use the triples 'as is', meaning the active branch is evaluated normally. Of course, the parties should not know which branch is inactive, so from the perspective of the parties it should appear plausible that either branch could have used randomized triples. To achieve this, for the active branch the parties also XOR masks onto the triples, but in this case each mask is a sharing of zero: hence the XORing is a no-op.

The problem of amortizing triples across branches thus reduces to the problem of generating secret shared masks, both uniform and 'all-zero'. We present techniques for efficiently generating these masks, the most general of which is based on oblivious vector-scalar multiplication, achieved by a small number of 1-out-of-2 oblivious transfers. The crucial point is that the protocols for generating masks require far less communication than protocols for generating triples.[1] Thus, we decrease communication and improve performance.

[1] We emphasize communication improvement because multiplication triples are often constructed from communication-expensive *oblivious transfers* (OTs). Silent

1.2 Advantage over [HKP20]

Recent work showed an improvement similar to ours: [HKP20] showed that oblivious transfers can be re-used across conditional branches in the classic GMW protocol (see Section 2.4 for a review).

However, [HKP20] has one significant disadvantage: their performance improvement depends on circuit topology. Efficient GMW implementations minimize latency by organizing circuits into *layers* of gates. The input wires into each layer are the outputs of previous layers only, and hence all gates in a particular layer can be executed simultaneously. This strategy yields latency proportional to the circuit's *multiplicative depth* instead of to the number of gates.

Due to this important optimization, [HKP20]'s performance improvement is limited by the 'relative alignment' of the layers across branches: two branches are highly aligned if each of their respective layers has a similar number of `AND` gates. Their protocol issues oblivious transfers that *simultaneously* run one gate per branch and hence cannot optimize gates that occur in different layers. The relative alignment of branches is dependent on the target application. [HKP20] suggests resorting to compiler technologies to extract more performance.

Our approach does not depend on topology. Instead, we depend only on the number of `AND` gates in each branch. The parties require enough triples to handle the maximum number of `AND` gates across the branches. It is difficult to analytically quantify our improvement over [HKP20] without a specific application in mind, but our experiments show that the improvement is significant. We ran both approaches across a variety of topologies, and on average we improved communication by 2.6× (see Section 8). In addition to concretely outperforming [HKP20], our approach also demonstrates that conditional improvement is possible for a different class of protocols (i.e. those based on triples), and hence is of independent interest.

1.3 Our Contributions

- **Efficient Re-use of Beaver Triples.** Our MPC protocol is secure against the semi-honest corruption of up to $p-1$ parties. The protocol re-uses triples across branches and requires a number of triples proportional only to the size of the longest execution path rather than to the size of the entire circuit.
- **Topology-Independent Improvement.** Unlike [HKP20], our improvement is independent of the topology of the conditional branches.
- **Implementation and evaluation.** We implemented our approach in `C++` and report performance (see Section 8). For 2PC and a circuit with 16

OT [BCG$^+$19] is an exciting new primitive that largely removes the communication overhead of OT. The trade off is increased computation: the classic OT extension of [IKNP03], which uses relatively little computation, is still preferable to Silent OT in most settings. This said, even if we were to use Silent OT, our improvement would be beneficial: we greatly reduce the needed number of OTs and hence would significantly reduce Silent OT computation overhead.

branches, we improve communication over state-of-the-art [HKP20] on average by 2.6× and over a standard triple-based protocol (i.e., without our conditional improvement) by 12×.

2 Preliminaries

2.1 Notation

- p denotes the number of parties.
- b denotes the number of branches.
- Subscript notation associates variables with parties. E.g., a_i is a variable held by party P_i.
- G denotes a pseudo-random generator (PRG).
- κ is the computational security parameter (e.g. 128).
- t denotes the 'taken' branch in a conditional, i.e. the branch that is active during the oblivious execution. $\bar{t}$ implies an inactive branch.
- Superscript notation associates variables with a particular branch. E.g., x^0 is associated with branch 0 while x^1 is associated with branch 1.
- $\in_\$$ denotes that the left hand side is uniformly drawn from the right hand set. E.g., $x \in_\$ \{0, 1\}$ denotes that x is a uniform bit.
- $\triangleq$ denotes that the left hand side is defined to be the right hand side.
- We manipulate vectors and bitstrings (i.e., vectors of bits):
  - Variables denoting vectors are indicated with bold notation, e.g. $\boldsymbol{a}$. If we wish to explicitly write out a vector, we use parenthesized, comma-separated values, e.g. $(a, b, ..., y, z)$.
  - We index vectors with brackets and use 1-based indexing, e.g. $\boldsymbol{a}[1]$.
  - When clear from context, n denotes vector length.
  - When two bitstrings are known to have the same length, we use $\oplus$ to denote the bitwise `XOR` sum:

$$\boldsymbol{a} \oplus \boldsymbol{b} = (\boldsymbol{a}[1], ..., \boldsymbol{a}[n]) \oplus (\boldsymbol{b}[1], ..., \boldsymbol{b}[n]) \triangleq (\boldsymbol{a}[1] \oplus \boldsymbol{b}[1], ..., \boldsymbol{a}[n] \oplus \boldsymbol{b}[n])$$

  - We indicate a bitwise vector scalar product by writing the scalar to the left of the vector:

$$a\boldsymbol{b} = a(\boldsymbol{b}[1], ..., \boldsymbol{b}[n]) \triangleq (a(\boldsymbol{b}[1]), ..., a(\boldsymbol{b}[n]))$$

- We manipulate `XOR` secret shares. Section 2.2 presents our secret share notation and reviews basic properties.

2.2 `XOR` Secret Shares

Our main contribution is a Beaver-triple based construction for efficient conditional branching. Additionally, we review prior work that is based on the classic GMW protocol. Both techniques are based on `XOR` secret shares. Thus, we briefly establish notation for `XOR` shares and review their properties.

An XOR secret sharing held amongst p parties is a vector of bits $(x_1, ..., x_p)$ where each party P_i holds x_i. We refer to the full vector as a *sharing* and to the individual bits held by parties as *shares*. The semantic value of a sharing (i.e., the cleartext value that the sharing represents) is the XOR sum of its shares. If the semantic value of a sharing $(x_1, ..., x_p)$ is a bit x, i.e. $x_1 \oplus ... \oplus x_p = x$, then we use the shorthand $\llbracket x \rrbracket$ to denote the sharing:

$$\llbracket x \rrbracket \triangleq (x_1, ..., x_p) \qquad\qquad \text{such that } x_1 \oplus ... \oplus x_p = x$$

Typically, sharings are used in a context where no strict subset of parties knows the semantic value of the sharing. Nevertheless, parties can easily perform homomorphic linear operations over XOR sharings.

- Parties XOR two sharings by locally XORing their respective shares:

$$\begin{aligned} \llbracket x \rrbracket \oplus \llbracket y \rrbracket &= (x_1, ..., x_p) \oplus (y_1, ..., y_p) && \text{Defn. sharing} \\ &= (x_1 \oplus y_1, ..., x_p \oplus y_p) && \text{Defn. vector } \texttt{XOR} \\ &= \llbracket x \oplus y \rrbracket && \texttt{XOR} \text{ commutes, assoc., defn. sharing} \end{aligned}$$

- Parties AND a sharing with a public constant by locally scaling each share.

$$\begin{aligned} c\llbracket x \rrbracket &= c(x_1, ..., x_p) && \text{Defn. sharing} \\ &= (cx_1, ..., cx_p) && \text{Defn. vector scalar product} \\ &= \llbracket cx \rrbracket && \texttt{AND} \text{ distributes over } \texttt{XOR}\text{, defn. sharing} \end{aligned}$$

- Parties encode public constants as sharings by letting P_1 take the constant as his/her share and letting all other parties take 0 as his/her share.

$$\llbracket c \rrbracket = (c, 0, ..., 0) \qquad\qquad \text{0 identity, defn. sharing}$$

  This allows the parties to XOR sharings with public constants.

A party can easily share her private bit x with the parties. She uniformly draws p bits, with the constraint that the p bits XOR to x. She then distributes $\llbracket x \rrbracket$ amongst the parties.

Parties can compute sharings of *uniform* values. To draw a uniform sharing, each party locally draws a uniform share. In our protocols, we overload $\in_\$$ notation to draw sharings: for example, $\llbracket x \rrbracket \in_\$ \{0, 1\}$ indicates that each party P_i draws a uniform share x_i.

Finally, parties can *reconstruct* the semantic value of a sharing. To do so, each party broadcasts his/her share (or sends it to a specified output party). Upon receiving all shares, each party locally XORs the shares.

For security, we often require that each party's share be uniformly chosen. We point out where shares are uniform when relevant.

`VS` Functionality:

- Input: Parties $P_1, ..., P_p$ input a scalar $\llbracket s \rrbracket$ and a bitstring $\llbracket \boldsymbol{x} \rrbracket$.
- Output: Parties output a uniform sharing $\llbracket s\boldsymbol{x} \rrbracket$.

Fig. 1: The functionality defining `VS` gate semantics. `VS` gates allow parties to multiply a sharing of a bitstring by a sharing of a scalar.

2.3 Vector Scalar Multiplication

Next, we review how parties operate non-linearly over sharings. In contrast to typical approaches that consider `AND` gates, we instead consider more general *vector scalar multiplication* gates, which we call `VS` gates. We consider these more expressive gates because they are needed to review prior work [HKP20] and because we use `VS` gates in our constructions.

To begin, we extend the notion of sharings to vectors. Specifically, we define the sharing of a vector to be a vector of sharings:

$$\llbracket \boldsymbol{x} \rrbracket = \llbracket (\boldsymbol{x}[1], ..., \boldsymbol{x}[n]) \rrbracket \triangleq (\llbracket \boldsymbol{x}[1] \rrbracket, ..., \llbracket \boldsymbol{x}[n] \rrbracket)$$

Suppose we wish to scale a shared vector $\boldsymbol{x}$ by a shared bit s. That is, we wish to compute the scalar product $\llbracket s\boldsymbol{x} \rrbracket$. Unlike linear operations, this vector scalar multiplication requires the parties to communicate.

[HKP20] showed that parties can use $p(p-1)$ oblivious transfers (OTs) to implement a `VS` gate. We review their `VS` protocol at a high level; Figure 1 specifies the protocol functionality. For simplicity, we focus on $p = 2$ parties and length-2 vectors, but the approach generalizes to arbitrary p and n.

Suppose two parties P_1, P_2, holding sharings $\llbracket s \rrbracket, \llbracket (a, b) \rrbracket$, wish to compute $\llbracket (sa, sb) \rrbracket$: semantically, they wish to scale the vector (a, b) by the bit s.

Observe the following equality over the desired semantic value:

$$\begin{aligned}
(sa, sb) &= s(a, b) && \text{Distribute} \\
&= (s_1 \oplus s_2)(a_1 \oplus a_2, b_1 \oplus b_2) && \text{Defn. sharing} \\
&= (s_1a_1 \oplus s_1a_2 \oplus s_2a_1 \oplus s_2a_2, s_1b_1 \oplus s_1b_2 \oplus s_2b_1 \oplus s_2b_2) && \text{Distribute} \\
&= s_1(a_1, b_1) \oplus s_1(a_2, b_2) \oplus s_2(a_1, b_1) \oplus s_2(a_2, b_2) && \text{Group}
\end{aligned}$$

The first and fourth summands can be computed locally by the respective parties. Thus, we need only show how to compute $s_1(a_2, b_2)$ (the remaining third summand is computed symmetrically). To compute this vector `AND`, the parties perform a single 1-out-of-2 OT of length-2 secrets. Here, P_2 plays the OT sender and P_1 the receiver. P_2 draws two uniform bits $x, y \in_\$ \{0, 1\}$ and allows P_1 to choose between the following two secrets:

$$(x, y) \qquad (x \oplus a_2, y \oplus b_2)$$

P_1 chooses based on s_1 and hence receives $(x \oplus s_1a_2, y \oplus s_1b_2)$. P_2 uses the vector (x, y) as her share of this summand. Thus, the parties hold $\llbracket s_1(a_2, b_2) \rrbracket$.

Put together, the full vector multiplication $s(a, b)$ uses only two 1-out-of-2 OTs of length-2 secrets. VS gates generalize to arbitrary numbers of parties and vector lengths: a vector scaling of n elements between p parties requires $p(p-1)$ 1-out-of-2 OTs of length-n secrets.

VS gates are important for our constructions. We present a modification to the above protocol, used once per conditional branch, that is optimized for scalar multiplication of *long* vectors (see Section 6.2). This modification is similar to techniques in [KK13, ALSZ13] and reduces communication by up to half.

2.4 Efficient Conditionals from VS Gates: [HKP20] Review

[HKP20] was the first work to significantly reduce the cost of branching in the multi-party setting. Their MOTIF protocol extends the classic GMW protocol with VS gates in order to amortize oblivious transfers across conditional branches. We review how VS gates enable this amortization.

For simplicity, consider two branches computed by two parties. Since the two branches are conditionally composed, one branch is active and one is inactive.

MOTIF's key invariant, set up by the protocol's circuit gadgets, is that on each wire of the inactive branch the parties hold a sharing $[\![0]\!]$, whereas on the active branch they hold valid sharings. XOR gates immediately propagate this invariant: on the inactive branch, XOR gates output $[\![0]\!]$, while on the active branch XOR gates output valid sharings.[2]

Next, we review how VS gates make use of and propagate the invariant. Let $[\![a^0]\!], [\![b^0]\!]$ be sharings held on wires in branch 0 and $[\![a^1]\!], [\![b^1]\!]$ be sharings held on wires in branch 1. Suppose the parties wish to compute both $[\![a^0 b^0]\!]$ and $[\![a^1 b^1]\!]$. Despite the fact that the parties compute two AND gates, they need only two 1-out-of-2 OTs. Let t denote the active branch. Hence, $a^{\bar{t}}$ and $b^{\bar{t}}$ are both 0.

Observe the following equalities:

$$\begin{aligned}(a^t \oplus a^{\bar{t}})b^t &= (a^t \oplus 0)b^t = a^t b^t \\ (a^t \oplus a^{\bar{t}})b^{\bar{t}} &= (a^t \oplus 0)0 = 0\end{aligned}$$

Thus computing both $[\![(a^t \oplus a^{\bar{t}})b^t]\!]$ and $[\![(a^t \oplus a^{\bar{t}})b^{\bar{t}}]\!]$, propagates the invariant: the active branch receives the correct sharing while the inactive branch receives $[\![0]\!]$. These products reduce to a vector-scalar product computable by a VS gate (see Figure 1).

$$[\![(a^t \oplus a^{\bar{t}})(b^t, b^{\bar{t}})]\!]$$

Thus, MOTIF computes two AND gates for the price of one. This improvement generalizes to an arbitrary number of branches.

[2] MOTIF does not natively support NOT gates because they would break the invariant: NOT maps $[\![0]\!]$ to $[\![1]\!]$. NOT gates can be implemented by XOR gates together with a distinguished wire that holds $[\![1]\!]$ on the active branch and $[\![0]\!]$ on the inactive branch.

`TripleGen` Functionality:

- Input: Parties $P_1, ..., P_p$ provide no input.
- Output: Let $a, b \in_\$ \{0, 1\}$ be uniform bits. Parties output uniform sharings $\llbracket a \rrbracket$, $\llbracket b \rrbracket$, and $\llbracket ab \rrbracket$.

Fig. 2: The Beaver Triple preprocessing functionality, `TripleGen`.

Branch Layer Alignment. As discussed in Section 1.2, the `MOTIF` protocol is dependent on circuit topology. The less *aligned* the layers of the branches are (branches that are highly aligned have similar numbers of `AND` gates in each layer), the less the circuit benefits from `MOTIF`.

In the above example, the parties issued two OTs to implement the two `AND` gates simultaneously. The parties can only perform this optimization if *inputs for both gates are available*. If not, the parties cannot amortize the OTs. Hence, gates in different layers cannot share OTs in layer-by-layer evaluation.

In the p-party protocol, in each layer `MOTIF` eliminates all OTs except for the total of $p(p-1) \cdot \max(w^i)$ OTs, where w^i is the number of `AND` gates in the current layer of branch i. In contrast, our technique does not depend on the circuit's topology and is always proportional only to the circuit's longest execution path.

2.5 Semi-honest Triple-Based Protocol Review

In this work, we amortize Beaver triples across conditional branches. We thus review how triples enable non-linear operations over `XOR` sharings.

Suppose the parties hold sharings $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ and wish to compute a uniform sharing $\llbracket xy \rrbracket$. Suppose further that the parties have a Beaver triple: they have three uniform sharings $\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket ab \rrbracket$ where $a, b \in_\$ \{0, 1\}$ are uniform bits unknown to any strict subset of parties. First, the parties locally compute $\llbracket a \oplus x \rrbracket$ and $\llbracket b \oplus y \rrbracket$, then reconstruct the semantic values $a \oplus x$ and $b \oplus y$ by broadcasting shares. This is secure: $a \oplus x$ leaks nothing about x because a is uniform and secret (and similarly for y). The parties can now compute $\llbracket xy \rrbracket$ as follows:

$$(a \oplus x)(b \oplus y) \oplus (a \oplus x)\llbracket b \rrbracket \oplus (b \oplus y)\llbracket a \rrbracket \oplus \llbracket ab \rrbracket = \llbracket xy \rrbracket$$

This protocol is simple and efficient: the parties broadcast only two bits per `AND` gate. However, because the triple values a and b are used as one-time-pads on semantic values, one triple is typically needed per gate. Thus, the parties must preprocess many triples according to the functionality in Figure 2. Computing this functionality is often the most expensive step in triple-based protocols. For example, triples might be achieved via the classic GMW protocol, requring $p(p-1)$ OTs per triple. In this work, we show a technique that re-uses triples across conditional branches and hence decreases overall cost.

3 Related Work

We review related work, focusing on works that optimize secure evaluation of conditional branches or that use multiplication triples in the malicious model.

MOTIF. The most closely related work is MOTIF [HKP20]. MOTIF amortizes oblivious transfers across conditional branches in the classic semi-honest GMW protocol [GMW87]. We reviewed this approach in detail in Section 2.4, explained why our approach outperforms MOTIF in Section 1.2, and present experimental comparisons between the two approaches in Section 8.

Stacked Garbling. Recent works demonstrated similar conditional improvements for the garbled circuit (GC) technique [Kol18, HK20b, HK20a]. [Kol18, HK20b] reduced communication in settings where one party knows which branch is active. [Kol18] is motivated by the use case where the GC generator knows the active branch, such as when evaluating one of several database queries. [HK20b] is motivated by zero knowledge proofs. [HK20a] superceded these prior works and for the first time showed that communication can be greatly improved even if *no party* knows which branch is active.

These works' "stacking" technique does not have an obvious analog for interactive multiparty protocols, so different techniques are needed, such as explored in [HKP20] and in this work. However, our approach follows the basic idea of *material re-use* introduced by Stacked Garbling: the expensive material is safely re-used in the (possibly incorrect) evaluation of inactive branches, whose output is obliviously discarded

Universal Circuits. We improve branching via cryptographic techniques. Another approach instead recompiles conditionals into a new form. Universal circuits (UCs) are programmable constructions that can evaluate any circuit up to a given size n. Branches can be compiled to one UC, potentially amortizing cost. At runtime, the UC can be programmed to compute the active branch.

Decades after Valiant's original work [Val76], UC enjoyed renewed interest due to its relevance to MPC, and UC constructions have steadily improved [KS08, LMS16, GKS17, AGKS19, KS16, ZYZL18]. Even with these improvements, representing conditional branches with UCs is often impractical. The state-of-the-art UC construction applied to a circuit with n gates still has factor $3 \log n$ overhead [LYZ$^+$20]. Thus, UC-based conditional evaluation is often more expensive than simply evaluating the condition naïvely. UC-based branching is superceded by cryptographic techniques such as Stacked Garbling, MOTIF, and this work.

Maliciously Secure Triple-Based Protocols. We present an improved triple-based semi-honest protocol. Two exciting and related lines of work explore triple-based protocols in the malicious model. These two lines differ primarily in how they preprocess triples. One line generates triples using homomorphic encryption

[BDOZ11, DPSZ12, DKL+13, KPR18] while another generates them using oblivious transfer [NNOB12, LOS14, FKOS15, KOS16, CDE+18]. To achieve malicious security, these methods rely on expensive primitives such as zero knowledge proofs and cut-and-choose. As a result, preprocessing is expensive.

Amortizing triples in these protocols would be an important improvement. While we make no claims in the malicious model, malicious improvements have historically been preceded by similar improvements in the semi-honest model. We leave investigating triple amortization in the malicious model as future work.

4 Technical Overview

As reviewed in Section 2.5, Beaver triples can efficiently and securely implement `AND` gates. In general, triples cannot be re-used, and hence a circuit with b branches each with n `AND` gates typically requires $n \cdot b$ triples.

As discussed in Section 1.1, our key observation is that triples *can* be re-used across conditional branches, as long as uniform `XOR` masks are *additionally* applied. These masks allow us to re-use the same triple to compute b gates across b branches. Thus b branches each with n gates require only n triples, improving the number of needed triples by factor b. Our technique does require the parties to hold additional shared per-branch masks, but these masks are computed cheaply.

This section presents our protocol, Π_{MT} (the 'masked triple protocol'), with detail sufficient to understand our contribution. Π_{MT} securely computes Boolean circuits among p parties, re-uses triples across conditional branches, and is secure against the semi-honest corruption of up to $p-1$ parties. Full formal algorithms, with accompanying proofs of correctness and security, are in Section 5.

4.1 Re-using Beaver Triples

For simplicity, consider only two branches, C^0 and C^1 and, without loss of generality, let C^0 be the active branch. The parties re-use the same set of triples for both branches. For the inactive branch, the parties will mask the triples with sharings of uniform bits; on the active branch the parties will mask the triples with sharings of zeros.

Suppose the parties hold sharings $[\![x^0]\!], [\![y^0]\!]$ on branch 0 and $[\![x^1]\!], [\![y^1]\!]$ on branch 1. Suppose further that they wish to obliviously compute one of $[\![x^0 y^0]\!]$ or $[\![x^1 y^1]\!]$, depending on which branch is active. Let $[\![a]\!], [\![b]\!], [\![ab]\!]$ be a uniform preprocessed triple. On the active branch, the parties mask $[\![a]\!]$ and $[\![b]\!]$ with uniform sharings of 0:

$$[\![a]\!] \oplus [\![0]\!] = [\![a]\!] \qquad [\![b]\!] \oplus [\![0]\!] = [\![b]\!]$$

The parties use this masked triple to compute branch 0's `AND` gate normally: the parties compute and reconstruct $a \oplus x^0$ and $b \oplus y^0$, and then locally compute the correct product:

$$(a \oplus x^0)(b \oplus y^0) \oplus (a \oplus x^0)[\![b]\!] \oplus (b \oplus y^0)[\![a]\!] \oplus [\![ab]\!] = [\![x^0 y^0]\!]$$

`MaskGen` Functionality:

- Parameters: The size of output bitstrings n.
- Input: Parties $P_1, ..., P_p$ provide no private input.
- Output: Let $s \in_\$ \{0,1\}$ be a uniform bit. Let $\boldsymbol{r} \in_\$ \{0,1\}^n$ be a uniform bitstring. Parties output $\llbracket s \rrbracket$ as well as the pair:

$$M^0, M^1 = \begin{cases} \llbracket 0^n \rrbracket, \llbracket \boldsymbol{r} \rrbracket & \text{if } s = 0 \\ \llbracket \boldsymbol{r} \rrbracket, \llbracket 0^n \rrbracket & \text{otherwise} \end{cases}$$

All output sharings are uniform.

Fig. 3: The `MaskGen` functionality provides parties with the pairs of masks needed to implement our optimization. The functionality computes two shared bitstrings. One bitstring is the all zero bitstring while the other is uniform. The two strings are swapped according to s, and the parties are given $\llbracket s \rrbracket$.

In contrast, on the inactive branch the parties mask their shares with uniform bits. Let $r, s \in_\$ \{0,1\}$ be two such bits and let the parties hold uniform sharings $\llbracket r \rrbracket$, $\llbracket s \rrbracket$. The parties compute:

$$\llbracket a \rrbracket \oplus \llbracket r \rrbracket = \llbracket a \oplus r \rrbracket \qquad \llbracket b \rrbracket \oplus \llbracket s \rrbracket = \llbracket b \oplus s \rrbracket$$

When the parties use this masked triple, they compute and reconstruct $a \oplus r \oplus x^1$ and $b \oplus s \oplus y^1$, and then locally compute the following expression:

$$(a \oplus r \oplus x^1)(b \oplus s \oplus y^1) \oplus (a \oplus r \oplus x^1)\llbracket b \oplus s \rrbracket \oplus (b \oplus s \oplus y^1)\llbracket a \oplus r \rrbracket \oplus \llbracket ab \rrbracket$$

The above expression does not correctly compute $\llbracket x^1 y^1 \rrbracket$, but this is irrelevant since all computations performed in the inactive branch are ultimately discarded by the circuit's conditional behavior.

Now, consider the security of the above re-use. As discussed above, each party views the following reconstructed semantic values:

$$a \oplus x^0 \quad b \oplus y^0 \quad a \oplus r \oplus x^1 \quad b \oplus s \oplus y^1$$

Because a, b, r, s are all uniform, this view is simulated by four uniform bits. Thus, our approach is secure. See Section 5.3 for a formal proof.

Although we have shown that mask sharings allow triple amortization, we have not discussed how these sharings are computed. Figure 3 formalizes `MaskGen`, a preprocessing functionality that computes strings of masks M^0 and M^1 such that (1) the parties receive a uniform sharing $\llbracket s \rrbracket$ where $s \in_\$ \{0,1\}$, (2) M^s is a uniform sharing of all zeros, and (3) $M^{\bar{s}}$ is a uniform sharing of random bits. During the preprocessing phase, the parties use `MaskGen` to preprocess strings with size sufficient to mask each triple. We formalize and prove secure $\mathit{\Pi}_{\mathsf{MT}}$ in the `MaskGen`-hybrid (and `TripleGen`-hybrid) model. Instantiations of `MaskGen` are provided and proved secure in Section 6.

Entering a conditional. `MaskGen` constructs two bitstrings that are ordered according to a uniform bit s (the parties hold a uniform sharing $[\![s]\!]$). To use our approach, the parties need to appropriately 'line up' the masks with the branches: the active branch should use the all zeros mask and the inactive branch should use the uniform mask. We assume the parties have explicit access to a sharing of the *branch condition*: the parties hold $[\![t]\!]$. Upon entering the conditional, the parties compute $[\![s \oplus t]\!]$ and then broadcast their shares to reconstruct $s \oplus t$. If $s \oplus t$ is 0, the parties do nothing. Otherwise, they locally swap their respective shares of the strings M^0 and M^1. After performing this conditional swap, the parties are assured that M^t is the all zeros mask and $M^{\bar{t}}$ is uniform. Note, $s \oplus t$ does not reveal the active branch t because s is uniform.

Exiting a conditional. Exiting conditionals is performed using ordinary Boolean logic. Let $[\![x^0]\!]$, $[\![x^1]\!]$ be corresponding output sharings from the two branches. We leave the branch by multiplexing each such pair of outputs: we compute $[\![x^0 \oplus t(x^0 \oplus x^1)]\!] = [\![x^t]\!]$. Thus, multiplexing requires one `AND` gate per conditional output.

4.2 Nested Branches

We have presented a technique for handling conditionals with only two branches. To generalize to higher branching factors, we *nest* conditionals. At each conditional, we use `MaskGen` to generate fresh masks and then apply these masks to the (possibly already masked) triples. This trivially and securely allows us to handle arbitrary branching control flow.

As a brief argument of security, consider that each branch uses a distinct mask string from each of its parent conditionals. Further, if the branch is inactive (1) at least one mask string will be uniform and (2) the `XOR` sum of all uniform mask strings for the branch is *unique*. Thus, all `AND` gate broadcasts can be simulated by uniform bits. We argue this more formally in Section 5.3.

We note that instead of nesting, it is possible to generalize our approach to directly handle vectors of conditionals, e.g., corresponding to program `switch` statements. This direction is not necessarily preferable: for a circuit with b branches, both techniques amortize a triple across up to b gates, and the work required to generate masks is very similar. We present the nested formalization due to its generality and relative simplicity.

5 Π_{MT}: Formalization and Proofs

We now present Π_{MT} formally. Section 5.1 begins by defining circuit syntax, including circuits with explicit conditional branching. We then specify our protocol in Section 5.2 and prove it correct and secure in Section 5.3.

5.1 Circuit Formal Syntax

Conditional branching is central to our approach. Thus, traditional circuits that include only low-level gates are insufficient for our formalization. We instead use the syntax of [HK20a] which makes explicit conditional branching. We review and formally present their syntax.

Conventionally, a circuit is a list of Boolean gates together with specified input and output wires. We refer to this representation as a *netlist*. We do not modify the semantics of netlists and evaluate them using the standard triple-based technique (see Section 2.5).

We extend the space of circuits with notion of a *conditional*. A conditional is parameterized over two circuits, C^0 and C^1. By convention, the first bit of input to the conditional is the branch condition t. The semantics of a conditional is that branch C^t is given the remaining input to the overall conditional, and C^t's output is returned.

Finally, we require an extra notion that allows us to place conditionals 'in the middle' of the overall circuit. A *sequence* is parameterized over two circuits C' and C''. When executed, the sequence passes its input to C', feeds the output of C' as input to C'', then returns the output of C''.

More formally, the space of circuits $\mathcal{C}$ is defined inductively. Let C^0, C^1, C', C'' be arbitrary circuits. The space of circuits is defined as follows:

$$\mathcal{C} \triangleq \mathit{Netlist}(\cdot) \mid \mathit{Cond}(C^0, C^1) \mid \mathit{Seq}(C', C'')$$

That is, a circuit is either a (1) netlist, a (2) conditional, or a (3) sequence. By arbitrarily nesting conditionals and sequences, we may achieve complex branching control structure.

5.2 Π_{MT} Formalization

Figure 4 presents our protocol for handling circuits with conditional branching. Π_{MT} first delegates to `TripleGen`, generating sufficient multiplication triples to handle the circuit, and then delegates to the sub-protocol `eval`. `eval` recursively walks the structure of the circuit and securely achieves circuit semantics.

Π_{MT} formalizes the ideas stated in Section 4 in a natural manner. The most interesting case in `eval` is the handling of conditionals, where we (1) invoke the `MaskGen` oracle, (2) mask the available triples, and (3) recursively evaluate both branches. Although we for clarity write `MaskGen` inline, the actual `MaskGen` protocol does not depend on any circuit values, and thus can be moved to a preprocessing phase. After evaluating both branches, we discard the inactive branch outputs and propagate the active branch outputs via a *multiplexer*. The multiplexer is implemented simply as a netlist, and computes the following function for each corresponding pair of branch outputs $[\![x^0]\!], [\![x^1]\!]$:

$$[\![x^0 \oplus t(x^0 \oplus x^1)]\!] = [\![x^t]\!] \qquad (1)$$

For simplicity, we abstract some algorithms and briefly describe them below. Other than $\Pi_{\mathtt{Base}}$, which is discussed in Section 2.5, we do not write these algorithms in full, as they are simple.

Functionality:

- Parameters: A circuit $C \in \mathcal{C}$.
- Input: Each party P_i inputs a private bitstring $\boldsymbol{inp}_i \in \{0,1\}^*$.
- Output: Each party P_i outputs $C(\boldsymbol{inp}_1, ..., \boldsymbol{inp}_p)$.

Semi-Honest Protocol:

```
$\mathit{\Pi}_{\mathsf{MT}}(C, \boldsymbol{inp}_1, ..., \boldsymbol{inp}_p)$ :
  $\llbracket \boldsymbol{inp} \rrbracket \leftarrow$ shareinput($\boldsymbol{inp}_1, ..., \boldsymbol{inp}_p$)
  $\llbracket \boldsymbol{triples} \rrbracket \leftarrow$ TripleGen(neededtriples($C$))
  $\llbracket \boldsymbol{out} \rrbracket \leftarrow$ eval($C, \llbracket \boldsymbol{triples} \rrbracket, \llbracket \boldsymbol{inp} \rrbracket$)
  return reconstruct($\llbracket \boldsymbol{out} \rrbracket$)


eval($C, \llbracket \boldsymbol{triples} \rrbracket, \llbracket \boldsymbol{inp} \rrbracket$) :
  switch $C$ :
    case Sequence($C', C''$) :
      ▷ Parse $\llbracket \boldsymbol{triples} \rrbracket$ into parts of sufficient length for each circuit.
      $\llbracket \boldsymbol{triples}' \rrbracket, \llbracket \boldsymbol{triples}'' \rrbracket \leftarrow \llbracket \boldsymbol{triples} \rrbracket$
      return eval($C'', \llbracket \boldsymbol{triples}'' \rrbracket$, eval($C', \llbracket \boldsymbol{triples}' \rrbracket, \llbracket \boldsymbol{inp} \rrbracket$))
    case Cond($C^0, C^1$) :
      ▷ Parse $\llbracket \boldsymbol{triples} \rrbracket$ into triples for the conditional and for the multiplexer.
      $\llbracket \boldsymbol{triples}_{cond} \rrbracket, \llbracket \boldsymbol{triples}_{mux} \rrbracket \leftarrow \llbracket \boldsymbol{triples} \rrbracket$
      ▷ Split the branch condition $t$ from the other branch inputs.
      ▷ By convention, the first input wire is the condition bit.
      $\llbracket t \rrbracket, \llbracket \boldsymbol{inp}_{cond} \rrbracket \leftarrow \llbracket \boldsymbol{inp} \rrbracket$
      $\llbracket M^0 \rrbracket, \llbracket M^1 \rrbracket, \llbracket s \rrbracket \leftarrow$ MaskGen($2 \cdot |\llbracket \boldsymbol{triples}_{cond} \rrbracket|$)
      $(s \oplus t) \leftarrow$ reconstruct($\llbracket s \rrbracket \oplus \llbracket t \rrbracket$)
      $(M^0, M^1) \leftarrow$ if $(s \oplus t)$ then $(M^1, M^0)$ else $(M^0, M^1)$
      $\llbracket \boldsymbol{out}^0 \rrbracket \leftarrow$ eval($C^0$, applymask($M^0, \llbracket \boldsymbol{triples}_{cond} \rrbracket$), $\llbracket \boldsymbol{inp}_{cond} \rrbracket$)
      $\llbracket \boldsymbol{out}^1 \rrbracket \leftarrow$ eval($C^1$, applymask($M^1, \llbracket \boldsymbol{triples}_{cond} \rrbracket$), $\llbracket \boldsymbol{inp}_{cond} \rrbracket$)
      return mux($\llbracket t \rrbracket, \llbracket \boldsymbol{out}^0 \rrbracket, \llbracket \boldsymbol{out}^1 \rrbracket, \llbracket \boldsymbol{triples}_{mux} \rrbracket$)
    case Netlist($g_1, ..., g_k$) :
      ▷ Evaluate with a standard triple-based protocol.
      return $\mathit{\Pi}_{\mathsf{Base}}((g_1, ..., g_k), \llbracket \boldsymbol{triples} \rrbracket, \llbracket \boldsymbol{inp} \rrbracket)$
```

Fig. 4: $\mathit{\Pi}_{\mathsf{MT}}$ allows p parties to securely compute a circuit $C \in \mathcal{C}$. $\mathit{\Pi}_{\mathsf{MT}}$ delegates to a recursive sub-procecure `eval`.

- $\mathit{\Pi}_{\mathsf{Base}}$ is the standard triple-based protocol as specified in Section 2.5. $\mathit{\Pi}_{\mathsf{Base}}$ takes as input (1) a vector of gates $(g_1, ..., g_k)$, (2) a vector of (possibly

masked) triples $[\![\boldsymbol{triples}]\!]$ and (3) the netlist input $[\![\boldsymbol{inp}]\!]$. $\Pi_{\texttt{Base}}$ returns a sharing of outputs $[\![\boldsymbol{out}]\!]$.
We emphasize that while we do not, for simplicity, explicitly list $\Pi_{\texttt{Base}}$, the protocol *is not* a black-box functionality.[3]
- `neededtriples` computes the number of needed triples for the circuit C. This computed number is equal to the number of `AND` gates on the circuit's longest execution path.
- `shareinput` allows the parties to construct and distribute sharings of their respective private inputs.
- `reconstruct` allows parties to reconstruct a sharing via broadcast.
- `mux` computes the per-output multiplexer function (Equation (1)).
- `applymask` specifies how mask sharings are `XOR`ed onto triples. Specifically, for each uniformly shared triple $[\![a]\!], [\![b]\!], [\![c]\!]$, we draw two bits from the mask sharing and `XOR` one bit onto both $[\![a]\!]$ and $[\![b]\!]$.

5.3 Π_{MT} Proofs

Now that we have introduced Π_{MT}, we prove it both correct and secure in the `MaskGen`- and `TripleGen`-hybrid model. We instantiate `MaskGen` in Section 6.

In both proofs, we refer to the notion of a *valid* triple. A triple is valid if it is uniformly shared and of the following form: $[\![a]\!], [\![b]\!], [\![ab]\!]$. That is, the third term is a share of the product of the first two terms. An *invalid* triple is a triple $[\![a]\!], [\![b]\!], [\![c]\!]$ such that $c \neq ab$. Invalid triples arise in our protocol due to the application of extra masks to the first two entries in triples.

Theorem 1 (Π_{MT} correctness). *For all circuits $C \in \mathcal{C}$ and all private inputs $\boldsymbol{inp}_1, ..., \boldsymbol{inp}_p \in \{0, 1\}^*$:*

$$\Pi_{\mathsf{MT}}(C, \boldsymbol{inp}_1, ..., \boldsymbol{inp}_p) = C(\boldsymbol{inp}_1, ..., \boldsymbol{inp}_p)$$

Proof. By induction on the structure of C. The inductive invariant is as follows:

> If the triples passed to `eval` are valid, then `eval` correctly implements the semantics of C.

We focus on conditionals; the correctness of netlists follows trivially from the standard triple-based protocol. The correctness of sequences is immediate.

Suppose C is a conditional $Cond(C^0, C^1)$. Further, suppose $[\![t]\!]$ is the branch condition. If the triples passed to the conditional are invalid, then the inductive invariant vacuously holds. Thus, we need only consider evaluation of a conditional on valid triples. The oracle call to `MaskGen` constructs two mask strings M^0, M^1 such that M^s is all-zero and $M^{\bar{s}}$ is uniform. By reconstructing $s \oplus t$

[3] We cannot support $\Pi_{\texttt{Base}}$ as a black-box because it is possible to implement a protocol that securely handles netlists, but that is insecure when given *masked* triples. For example, the parties could out-of-band check the validity of triples and reveal all inputs if the triples are found invalid. Therefore, $\Pi_{\texttt{Base}}$ is a white-box protocol.

and accordingly locally swapping the two mask strings, the parties ensure M^t is the all-zero string. Thus, C^t is given valid triples (the valid triples are masked by sharings of zeros and hence remain valid) and, by induction, returns the correct semantic outputs. $C^{\bar{t}}$ will not return correct values, but these outputs are discarded by the multiplexer. Thus, conditionals support the inductive invariant.

The top level circuit is given valid triples via the oracle call to `TripleGen`. This fact, combined with the inductive invariant, implies that Π_{MT} is correct. □

Theorem 2 (**Π_{MT} security**). *Π_{MT} is secure against semi-honest corruption of up to $p-1$ parties in the* `TripleGen`*-hybrid and* `MaskGen`*-hybrid model.*

Proof. By construction of a simulator for one party, which we later generalize to simulate up to $p-1$ parties. Each broadcast received by a party can be simulated by a uniform bit.[4] We prove this simulation secure by induction on the structure of the circuit C. The inductive invariant is as follows:

> Let $[\![a]\!], [\![b]\!], [\![c]\!]$ be a (possibly invalid) triple. For each triple, we refer to the semantic values a and b as the *one-time-pad* parts. `eval` uses both one-time-pad parts of each triple to mask at most one cleartext value.

For netlists, this is trivial: we use a distinct triple for each `AND` gate, and each one-time-pad part is used only to mask one of the gate inputs. Similarly, sequences satisfy the inductive invariant trivially: we provide different triples to both parts of the sequence.

Therefore we focus on conditionals. Consider a conditional $Cond(C^0, C^1)$. As a brief aside from proving that the inductive invariant holds, while the parties reconstruct $s \oplus t$, s is uniform, and hence this leaks nothing about t (i.e., $s \oplus t$ can be simulated by a uniform bit). Now, returning to the invariant: We first split the triples into sufficient numbers for the conditional body and for the multiplexer. The multiplexer is implemented by a netlist, and hence trivially satisfies our invariant. The conditional body is more complicated. Indeed, we use the same triples to evaluate both branches. However, our call to `MaskGen` together with the conditional swap ensures that $M^{\bar{t}}$ is a sharing of uniform bits. When we apply $M^{\bar{t}}$ to the triples, we re-randomize the one-time-pad parts of the triples. (Note, applying M^t (the all zeros mask) has no effect on the one-time-pad parts.) Thus, we provide independent one-time-pad parts to both C^0 and C^1, satisfying the inductive invariant.

Because each one-time-pad part is (1) uniform and (2) used to mask at most one cleartext value, and because each broadcast is masked by a one-time-pad part, each broadcast can be simulated by a uniform bit. Thus, we can simulate a single party's view.

The generalization from simulating one party to simulating up to $p-1$ is based on a simple observation about `XOR` secret shares: the view of $p-1$ parties

[4] One caveat is that broadcasts used to reconstruct the circuit's outputs must `XOR` to the correct output value. The simulator must arrange the simulated output broadcasts such that they appropriately add up. This is typical in MPC proofs and is easy to set up.

holds no more information than the share of 1 party. The remaining broadcasts from the remaining, unsimulated parties can be simulated by uniform bits.

$\mathit{\Pi}_{\mathsf{MT}}$ is secure against semi-honest corruption of up to $p-1$ parties.

□

Some MPC techniques, e.g., computing multiplicative inverse [BIB89], rely on opening (randomized) intermediate values. This may not always be compatible with our optimization, since our randomization of the inactive branch may cause an invalid opened value, thereby revealing that it was in fact inactive.

6 Semi-Honest `MaskGen` Instantiations

In this section, we instantiate `MaskGen` (Figure 3). We present three protocols, two formally and one informally, that follow two general approaches:

1. The first approach is generic in that it works for an arbitrary number of parties and is based on vector scalar multiplication (Section 2.3). Since our approach often uses long masks, we introduce a useful trick that improves vector scalar multiplication for long vectors.
2. In special cases, masks can be more efficiently derived starting from short seeds. We present two and three-party protocols which require communication proportional only to κ rather than to the mask length n.

6.1 *p*-Party Mask Generation

Our general mask generation technique, `Π-MaskGen-VS` (Figure 5), allows p parties to preprocess length-n masks using only a single `VS` gate.

In this protocol, parties jointly sample a uniform sharing of a uniform bit $[\![s]\!]$ and a uniform bitstring $[\![\boldsymbol{r}]\!]$. The parties compute $[\![s\boldsymbol{r}]\!]$ via a `VS` gate, set the first mask to $[\![M^0]\!] = [\![s\boldsymbol{r}]\!]$, and set the second mask to $[\![M^1]\!] = [\![s\boldsymbol{r}]\!] \oplus [\![\boldsymbol{r}]\!]$.

`Π-MaskGen-VS` is correct and secure.

Theorem 3. `Π-MaskGen-VS` *correctly implements* `MaskGen`*.*

Proof. s and $\boldsymbol{r}$ are uniform. Depending on s, the product $s\boldsymbol{r}$ is, of course, either all zeros or $\boldsymbol{r}$. Thus, setting $[\![M^0]\!] = [\![s\boldsymbol{r}]\!]$ and $[\![M^1]\!] = [\![s\boldsymbol{r}]\!] \oplus [\![\boldsymbol{r}]\!]$ places the all zeros mask in M^s. □

Theorem 4. `Π-MaskGen-VS` *is secure against semi-honest corruption of up to $p-1$ parties in the* `VS`*-hybrid model.*

Proof. We communicate only once: when evaluating a single `VS` gate. Hence, the simulator is trivially constructed from the `VS` gate simulator. □

Functionality:

- Parties $P_1, \ldots, P_p$ compute `MaskGen`.

Semi-Honest Protocol:

```
▷ $n$ is the output mask length.
Π-MaskGen-VS($n$) :
  ▷ Parties jointy sample $\boldsymbol{r}$ and $s$.
  $[\![\boldsymbol{r}]\!] \in_{\$} \{0,1\}^n$
  $[\![s]\!] \in_{\$} \{0,1\}$
  $[\![s\boldsymbol{r}]\!] \leftarrow [\![s]\!][\![\boldsymbol{r}]\!]$   ▷ Computed via VS gate.
  $[\![M^0]\!] \leftarrow [\![s\boldsymbol{r}]\!]$
  $[\![M^1]\!] \leftarrow [\![s\boldsymbol{r}]\!] \oplus [\![\boldsymbol{r}]\!]$
  ▷ $M^s$ is all zeros, $M^{\bar{s}}$ is uniform.
  return $[\![s]\!]$, $[\![M^0]\!]$, $[\![M^1]\!]$
```

Fig. 5: Protocol `Π-MaskGen-VS` is our default method for generating masks, and is secure for an arbitrary number of parties.

Vector Scalar Multiplication for Long Vectors. We have shown that `VS` gates can efficiently compute pairs of masks. However, this requires us to evaluate `VS` gates over potentially long vectors: we compute `VS` gates over vectors with length proportional to the number of `AND` gates, which can be arbitrarily high.

As discussed in Section 2.3, we decompose vector scalar products into summands, some that are computed locally and others that are computed interactively. For each interactive summand, one party holds a bit a, one a vector $\boldsymbol{b}$, and the two must jointly compute $[\![a\boldsymbol{b}]\!]$. Let n be the length of $\boldsymbol{b}$. To compute this product, the protocol presented by [HKP20] requires *two* messages of length n. In this section, we introduce a natural trick, `Π-Half-VS-Long`, that reduces this communication cost by half: only one message of length n need be sent, and the other can be derived from a pseudo-random seed. Both the functionality and the protocol are listed in Figure 6. We explain the `Π-Half-VS-Long` trick in more detail in our proof of correctness. Our trick is similar to techniques in [KK13, ALSZ13]. Recall (from Section 2.3) that $p(p-1)$ interactive summands emerge from a single vector scalar multiplication. Thus, we can compose a full vector scalar multiplication protocol from $p(p-1)$ calls to `Π-Half-VS-Long`. We refer to this full protocol as `Π-VS-Long`.

Theorem 5. *`Π-Half-VS-Long`, and hence `Π-VS-Long`, is correct.*

Proof. The key observation is that P_1's input bit a determines one of two possible outcomes for the vector scalar multiplication. If $a = 0$, the output is a sharing of all zeros. In this case, P_1 and P_2's output shares must `XOR` to zeros. If $a = 1$, the output is a sharing of P_2's input vector $\boldsymbol{b}$.

Functionality:

- INPUT: Party P_1 inputs a private bit a, P_2 inputs a private vector $\boldsymbol{b}$.
- OUTPUT: Parties output a uniform sharing $[\![a\boldsymbol{b}]\!]$.

Semi-Honest Protocol:

```
$\Pi$-Half-VS-Long($a, \boldsymbol{b}$) :
$P_1$ :                                                $P_2$ :
                                                       $\boldsymbol{S} \in_\$ \{0,1\}^\kappa$
                                                       ▷ $P_2$ expands $\boldsymbol{S}$ to get his share of $a\boldsymbol{b}$.
                                                       $(a\boldsymbol{b})_2 \leftarrow G(\boldsymbol{S})$
                                                       $\boldsymbol{k} \in_\$ \{0,1\}^\kappa$
                                                       ▷ $P_2$ sends $G(\boldsymbol{k}) \oplus ((a\boldsymbol{b})_2 \oplus \boldsymbol{b})$ to $P_1$.
                                                       $\boldsymbol{v} \leftarrow G(\boldsymbol{k}) \oplus ((a\boldsymbol{b})_2 \oplus \boldsymbol{b})$
  $\boldsymbol{v} \leftarrow$ recv($P_2$)                             send($\boldsymbol{v}, P_1$)
  ▷ $P_1$ and $P_2$ run OT.
  ▷ $P_1$ is the OT receiver.                           ▷ $P_2$ is the OT sender.
  $\boldsymbol{w} \leftarrow \mathtt{OT}_{\mathtt{recv}}(a)$                               $\mathtt{OT}_{\mathtt{send}}(\boldsymbol{S}, \boldsymbol{k})$
  ▷ $\boldsymbol{w} = a(\boldsymbol{S} \oplus \boldsymbol{k}) \oplus \boldsymbol{S}$.
  ▷ If $a = 0$, $G(\boldsymbol{w}) = G(\boldsymbol{S}) = (a\boldsymbol{b})_2$.
  ▷ If $a = 1$, $G(\boldsymbol{w}) \oplus \boldsymbol{v} = G(\boldsymbol{k}) \oplus \boldsymbol{v}$.
                               $= (a\boldsymbol{b})_2 \oplus \boldsymbol{b}$.
  $(a\boldsymbol{b})_1 \leftarrow G(\boldsymbol{w}) \oplus a\boldsymbol{v}$
  return $(a\boldsymbol{b})_1$                                    return $(a\boldsymbol{b})_2$
```

Fig. 6: Π-Half-VS-Long can be used to evaluate the interactive subterms that emerge from computing a VS gate.

We achieve this functionality with a single 1-out-of-2 OT of length κ strings. P_1 acts as the OT receiver and uses as her choice bit a. If $a = 0$, P_1 receives the seed that P_2 used to generate her share. P_1 expands this seed and obtains the same share as P_2. If $a = 1$, P_1 receives a key that helps him to decrypt a ciphertext sent separately by P_2. The ciphertext holds a valid share of $\boldsymbol{b}$.

The correctness of Π-VS-Long is immediate from the correctness of Π-Half-VS-Long and [HKP20]'s VS instantiation. □

Next, we prove this faster vector scalar multiplication procedure secure. Ideally, we would modularly prove Π-Half-VS-Long and Π-VS-Long secure by simulation. Unfortunately, this is not possible. Specifically, suppose that in Π-Half-VS-Long P_1 provides input $a = 0$. In this case, P_1 outputs the expansion of the pseudorandom seed $\boldsymbol{w}$ received by the OT oracle. Now we need to simulate $\boldsymbol{w}$ that matches the expansion $G(\boldsymbol{w})$ output by the protocol. Since G

is assumed secure, this simulation is infeasible. Therefore, we forego modularity, and instead prove the security of our top level circuit protocol, Π_{MT}, but where we instantiate the `MaskGen` functionality based on `Π-VS-Long`. With our PRG-utilizing subprocedures 'inlined', we can prove the top-level protocol secure, since the expansions of PRG seeds no longer appear as protocol outputs and we can simulate the seeds simply by random strings.

Theorem 6. *Let* `Π-MaskGen-VS′` *be the protocol* `Π-MaskGen-VS` *(Figure 5), with* `VS` *instantiated by* `Π-VS-Long`. *Let* Π'_{MT} *be the protocol* Π_{MT} *(Figure 4), with* `MaskGen` *instantiated by* `Π-MaskGen-VS′`. Π'_{MT} *is secure against semi-honest corruption of up to* $p-1$ *parties in the* `TripleGen`*-hybrid and OT-hybrid model.*

Proof. By construction of a simulator.

The proof is similar to that of Theorem 2, so we elide most details. Because we explicitly instantiate `MaskGen` with `Π-MaskGen-VS′`, we focus on the corresponding difference in the proof and explain how we simulate `Π-MaskGen-VS′` messages. Namely, we argue that all messages of `Π-MaskGen-VS′` are simulated by uniform bits.

`Π-MaskGen-VS′` invokes `Π-VS-Long`, which in turn makes $2(p-1)$ per-party calls to `Π-Half-VS-Long`. Each pair of parties P_i, P_j jointly call `Π-Half-VS-Long` twice, once where P_i is the receiver and once where P_i is the sender.

When P_i is the receiver, he receives two messages:

- First, P_i receives from P_j an encrypted share of $\boldsymbol{b}$: P_j chooses a PRG seed $\boldsymbol{k}$, expands $G(\boldsymbol{k})$, and then sends $G(\boldsymbol{k}) \oplus G(\boldsymbol{S}) \oplus \boldsymbol{b}$ to P_i. The simulator can simulate this received message by uniform bits because $\boldsymbol{k}$ and $\boldsymbol{S}$ are both uniform and because G is a secure PRG.
- Second, P_i receives a message from the OT oracle. Depending on its input bit a_i, P_i receives either the seed $\boldsymbol{k}$ or the seed $\boldsymbol{S}$ (which was used to generate $(a\boldsymbol{b})_2$). In either case, $\boldsymbol{k}$ and $\boldsymbol{S}$ are simulated by a uniform string. This does not conflict with the previously simulated message $G(\boldsymbol{k}) \oplus G(\boldsymbol{S}) \oplus \boldsymbol{b}$, since one of the seeds $\boldsymbol{k}$ or $\boldsymbol{S}$ remains hidden from P_i.

If P_i is the sender, no message is received; P_i's view is trivially simulated.

It is easy to see that the masks produced by `Π-MaskGen-VS′` are used exactly once in Π'_{MT}, and hence the inductive invariant of Theorem 2 is maintained.

Π'_{MT} is secure. □

6.2 Efficient 2PC and 3PC Mask Generation

In this section, we present two efficient implementations of `MaskGen`, one for two parties, and one for three. At a high level, these methods are based on (1) distributing pseudo-random seeds and (2) expanding the seeds with a PRG into n-bit masks. The advantage of these seed-based methods is that they use communication proportional only to κ. This is a significant improvement over `Π-MaskGen-VS`, which uses communication proportional to the mask length n.

Functionality:

- Parties P_1 and P_2 compute MaskGen.

Semi-Honest Protocol:

```
Π-MaskGen-2P($n$) :
  $[\![s]\!] \in_\$ \{0,1\}$ ; $[\![\boldsymbol{S}]\!] \in_\$ \{0,1\}^\kappa$
  $[\![s\boldsymbol{S}]\!] \leftarrow [\![s]\!][\![\boldsymbol{S}]\!]$   ▷ Computed via VS gate.
  ▷ $P_1, P_2$ compute uniform shares of two seeds $\boldsymbol{S}^0, \boldsymbol{S}^1$.
  $[\![\boldsymbol{S}^0]\!] \leftarrow [\![s\boldsymbol{S}]\!]$ ; $[\![\boldsymbol{S}^1]\!] \leftarrow [\![s\boldsymbol{S}]\!] \oplus [\![\boldsymbol{S}]\!]$
  ▷ $P_1, P_2$ expand their PRG seeds into $n$-bit masks.
  $[\![M^0]\!] \leftarrow$ Π-Expand-2P($[\![\boldsymbol{S}^0]\!], n$)
  $[\![M^1]\!] \leftarrow$ Π-Expand-2P($[\![\boldsymbol{S}^1]\!], n$)
  ▷ $M^s$ is all zeros, $M^{\bar{s}}$ is uniform.
  return $[\![s]\!]$, $[\![M^0]\!]$, $[\![M^1]\!]$

▷ $P_1$ and $P_2$ expand their respective seeds into $n$-bit strings.
Π-Expand-2P($(\boldsymbol{S}_1, \boldsymbol{S}_2), n$) :
  $P_1$ : return $G(\boldsymbol{S}_1, n)$                    $P_2$ : return $G(\boldsymbol{S}_2, n)$
```

Fig. 7: Π-MaskGen-2P is an efficient two-party protocol for generating masks.

Two party improved protocol. Figure 7 presents our protocol for two parties, Π_{MT}-2P. Here, the parties use vector scalar multiplication to distribute XOR sharings of two length-κ strings; one sharing encodes a uniform string and one encodes the all zeros string. The parties then interpret their respective shares as PRG seeds and apply G. Because of the nature of XOR sharings, this means that for the all zeros sharing, the parties generate the same pseudorandom expansion, so the resultant expansions are a sharings of all zeros. In contrast, the expansion of the random sharing leads to a larger pseudorandom sharing.

By using this protocol, the two parties can share arbitrarily long masks at the cost of only $O(\kappa)$ bits of communication.

Theorem 7. Π-MaskGen-2P *correctly implements* MaskGen.

Proof. By the correctness of VS gates and properties of XOR shares.

One of $[\![\boldsymbol{S}^0]\!]$ and $[\![\boldsymbol{S}^1]\!]$ is a sharing of zeros while the other is a sharing of a random bitstring. The position of the all-zeros sharing is determined by a uniform bit s. Consider one such sharing, and interpret both shares as PRG seeds. If the parties' two seeds are the same, then the expanded masks will also be the same, and will therefore XOR to zeros. If the parties' two seeds differ, then, by the properties of the PRG, the expanded masks will XOR to a uniform value.

Π-MaskGen-2P is correct. □

We next prove $\mathtt{\Pi\text{-}MaskGen\text{-}2P}$ secure. Like $\mathtt{\Pi\text{-}Half\text{-}VS\text{-}Long}$, we unfortunately cannot modularly prove this protocol secure by simulation: each party outputs the expansion of a PRG seed that appears in the party's view. We therefore instead prove that Π_{MT} is secure in the case where we instantiate `MaskGen` with $\mathtt{\Pi\text{-}MaskGen\text{-}2P}$. This higher level approach works because the output of a PRG does not appear as final output, so the PRG seeds can be simulated.

Theorem 8 ($\Pi_{\mathsf{MT}}\text{-}\mathtt{2P}$ Security). *Let $\Pi_{\mathsf{MT}}\text{-}\mathtt{2P}$ be Π_{MT} (Figure 4), where we instantiate* `MaskGen` *with* $\mathtt{\Pi\text{-}MaskGen\text{-}2P}$*. $\Pi_{\mathsf{MT}}\text{-}\mathtt{2P}$ is secure against semi-honest corruption of a single party in the* `TripleGen`*-hybrid and* `VS`*-hybrid model.*

Proof. The proof is nearly identical to that of Π_{MT} (Theorem 2); we therefore focus our discussion on the call to $\mathtt{\Pi\text{-}MaskGen\text{-}2P}$.

In $\mathtt{\Pi\text{-}MaskGen\text{-}2P}$, the parties jointly sample uniform sharings $[\![\boldsymbol{s}]\!]$ and $[\![\boldsymbol{S}]\!]$, and then compute $[\![s\boldsymbol{S}]\!]$ via `VS`. `VS` outputs uniform sharings, and so the message each party receives from the `VS` oracle is simulated by uniform bits. The parties locally expand their shares to obtain masks $[\![M^0]\!]$ and $[\![M^1]\!]$. Because G is a secure PRG, $[\![M^{\bar{s}}]\!]$ is a sharing of a *uniform string.*

Now, recall Theorem 2's inductive invariant: we must ensure that the top-level protocol uses the one-time-pad part of each multiplication triple at most once. $\Pi_{\mathsf{MT}}\text{-}\mathtt{2P}$ XORs the current triples with both $[\![M^0]\!]$ and $[\![M^1]\!]$. Because $[\![M^{\bar{s}}]\!]$ is a sharing of a uniform string, this appropriately rerandomizes the triples into the inactive branch, and hence we support the inductive invariant.

$\Pi_{\mathsf{MT}}\text{-}\mathtt{2P}$ is secure against semi-honest corruption of a single party. □

Three party informal `MaskGen` protocol. The three party efficient `MaskGen` protocol is a relatively straightforward generalization of the two party protocol. However, the mask generation is notationally complex, so for simplicity we present informally. A similar technique was used in [BKKO20] to help to construct a two-private three-server distributed point function.

Unlike the two-party protocol, P_1, P_2, and P_3 each obtain two *pairs* of seeds. Each pair is used to generate one mask by (1) expanding both seeds with a PRG into an n-bit string and (2) `XOR`ing the two expanded outputs together. At a high level, we ensure that:

1. For the all zeros mask, each party holds the same seed as one other party. Thus, their PRG expansions `XOR` to zeros.
2. For the uniform mask, each party holds a seed distinct from all other parties. Thus, their PRG expansions `XOR` to a uniform mask.

The key difficulty is in making the two above scenarios indistinguishable from the perspective of any strict subset of parties. We contrast these two scenarios, showing that they appear indistinguishable.

In the first case, the parties are given seeds as follows:

$$P_1 : \boldsymbol{S}_1, \boldsymbol{S}_2 \qquad P_2 : \boldsymbol{S}_1, \boldsymbol{S}_3 \qquad P_3 : \boldsymbol{S}_2, \boldsymbol{S}_3$$

If we consider an adversary who corrupts any two parties, he will see that one seed is shared between them and the others appear uniform.

In the second case the parties are given seeds as follows:

$$P_1 : \boldsymbol{S}_4, \boldsymbol{S}_5 \qquad P_2 : \boldsymbol{S}_4, \boldsymbol{S}_6 \qquad P_3 : \boldsymbol{S}_4, \boldsymbol{S}_7$$

As in the first case, an adversary that corrupts two parties sees one seed in common; the others are uniform. Hence, the cases are indistinguishable for any one or two parties.

Thus, Π_{MT} instantiated with this three party `MaskGen` trick results in a secure and correct protocol. Seed distribution can easily be implemented by GMW extended with `VS` gates: the parties sample seven uniform seeds $\boldsymbol{S}_1, ..., \boldsymbol{S}_7 \in_\$ \{0,1\}^\kappa$, swap them using a `VS` gate, and output each of them to the appropriate party.

7 Implementation

We implemented our approach in `C++`. Specifically, we implemented Π_{MT}, instantiating `MaskGen` with both `Π-MaskGen-VS` and `Π-MaskGen-2P` (we did not implement the three-party variant). We instantiated `TripleGen` with the natural approach based on random OT. For comparison, we also (1) implemented a standard triple-based protocol and (2) incorporated `MOTIF`'s implementation into our repository. We discuss key aspects of our implementation in Section 7.1.

To the best of our knowledge, there is no comprehensive suite of MPC benchmark circuits, particularly for circuits that include conditional branches. Thus, we implemented a random circuit generator to produce benchmarks. In designing the circuit generator, our key goal was to capture the impact of branch alignment on `MOTIF`'s performance such that we can highlight our improvement. The circuit generator samples circuits with a variety of branch alignments. We describe details of circuit generation in Section 7.2.

7.1 Key Implementation Aspects

Our implementation of Π_{MT} is straightforward, but we note some of its interesting aspects. We use the 1-out-of-2 OT protocol of [IKNP03] as implemented by EMP [WMK16] in order to generate both triples and masks. In Π_{MT} and standard triple-based protocol, we list `AND` gates in layers so that we can parallelize broadcasts for `AND`s in the same circuit layer. `MOTIF` similarly parallelizes OTs for `VS` gates in the same layer. Thus, all three protocols use communication rounds proportional to the circuit's multiplicative depth.

7.2 Random Circuit Generation

Circuit generation consists of three main steps:

1. We parameterize circuits on the numbers of conditional branches, the number of circuit layers, the number of XOR and AND gates per branch, and the number of input/output wires to each branch. Each branch uses the same parameters.
2. We uniformly assign a number of gates to each branch's layers. We implement this functionality with a RANCOM algorithm [NW78], which is based on the balls-in-cells problem and is called separately for each branch.
3. We connect the gates layer by layer. Specifically, we maintain a pool of wires whose value has already been assigned (i.e., it is a branch input or the output of a gate). For each gate, we uniformly sample two inputs from the pool and choose a fresh output wire. Once a layer has been entirely connected, we add all of that layer's gate outputs to the pool.

The above strategy is relatively ad-hoc, and may not be representative of all applications. Again, we adopt the above approach (1) to show the impact of circuit alignment on our relative performance over MOTIF and (2) because no standard benchmark suite exists.

8 Performance Evaluation

We compare $\mathit{\Pi}_{\mathsf{MT}}$ to MOTIF and the standard triple-based protocol. We compare these protocols for various numbers of parties. All experiments were run on a commodity laptop running Ubuntu 20.04 with an Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz and 16GB RAM. All parties were run on the same machine and network settings were configured with the `tc` command. We averaged each data point over 100 runs.

In each experiment, we generated random circuits as described in Section 7.2. We fixed the circuit parameters to 10 layers, $30,000$ AND gates per branch and $30,000$ XOR gates per branch. We set the number of branch input and output wires to 128. We generated a new circuit with these same parameters for each run of each experiment. We performed and report on three experiments:

1. We fixed the number of branches to two, fixed the number of parties to two, and explore variation in performance based on branch alignment (Section 8.1).
2. We fixed the number of parties to two, varied the number of branches, and explore corresponding communication and wall-clock runtime (Section 8.2 and Section 8.3).
3. We fixed the branching factor to 16, varied the number of parties, and explore corresponding communication.

Each experiment shows that our approach is preferred in almost every setting.

8.1 Branch Alignment

We first demonstrate MOTIF's dependence on circuit topology in the case of two branches. Figure 8 plots the distribution of the number of random OTs needed for

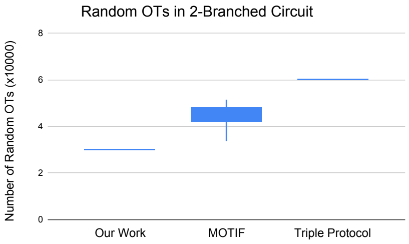

Fig. 8: Random OTs required to evaluate a circuit with two branches.

two parties to evaluate each protocol. Across all 100 runs, Π_{MT} and the standard triple-based protocol always need the same number of OTs. On the other hand, `MOTIF`'s performance differs depending on branch alignment. Because we sample alignments uniformly, this results in an increased number of consumed OTs.

Discussion. For two branches and on average, our approach required 1.5× fewer OTs than `MOTIF` and consistently required 2× fewer OTs than the standard triple-based protocol. Given that random OTs are the main communication bandwidth bottleneck, `MOTIF` is far from reducing communication by the optimal factor 2×. Π_{MT} never used more OTs than `MOTIF`. `MOTIF`'s best run required 1.12× more OTs than Π_{MT} and 1.71× in the worst case.

8.2 Communication

We next report our 2PC communication improvement over both `MOTIF` and the standard triple-based protocol as a function of branching factor. We instantiated `MaskGen` with `Π-MaskGen-2P`.

Figure 9 plots both preprocessing communication and total communication. For further reference, Figure 10 tabulates our communication improvement.

In our measurements, preprocessing constitutes both triple generation and mask generation. Each data point is averaged over 100 runs; the amount of communication may differ from run to run because each circuit has a randomly generated topology. In Π_{MT} the total communication is constant. In contrast, `MOTIF` communication differs significantly across runs due to the layering issue explained in Section 2.4.

Discussion. In this metric, Π_{MT} is preferred:

- **Preprocessing Communication.** On 16 branches, we improve communication by 2.96× over `MOTIF` and by 14.4× over the standard triple-based protocol. There are three reasons we did not achieve 16× improvement over standard triple-based protocol. First, both the standard triple-based approach and ours must perform the same number of *base OTs* to set up

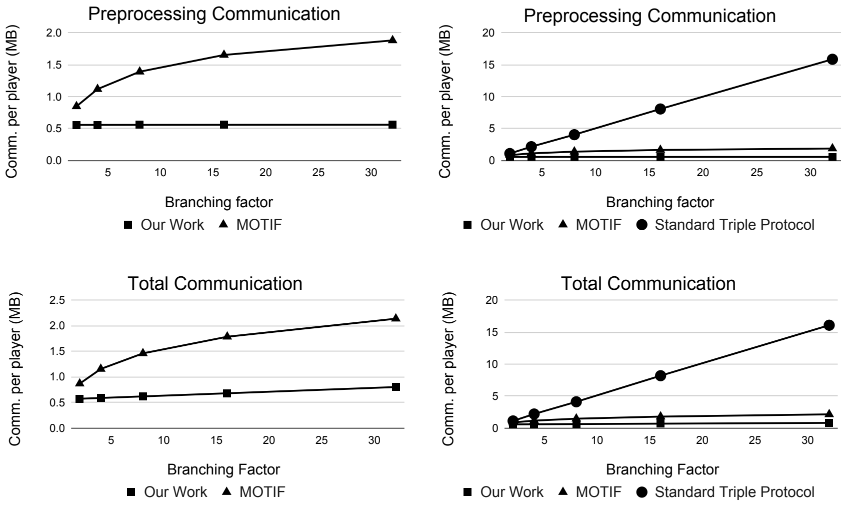

Fig. 9: 2PC comparison of $\mathit{\Pi}_{\mathsf{MT}}$ against `MOTIF` and a standard triple-based protocol. We plot the following metrics as functions of the *branching factor*: the preprocessing per-party communication (top), the total per-party communication (bottom).

# Branches	$\mathit{\Pi}_{\mathsf{MT}}$**(MB)**	`Π-MOTIF`**(MB)**	**Broadcast (MB)**
2	0.57	0.86	1.11
4	0.58	1.16	2.19
8	0.61	1.46	4.10
16	0.67	1.79	8.19
32	0.80	2.14	16.09

Fig. 10: Per-party communication improvement for our 2PC random circuit experiment as a function of the branching factor.

an OT extension matrix [IKNP03]. This adds a small amount of communication (around 20KB) common to both approaches, which cuts slightly into our advantage. Second, we need one OT per each of $b-1$ mask pairs. Third, entering and exiting conditionals have very small overhead differences.

- **Total Communication.** On 16 branches, our approach improves total communication by 2.6× over `MOTIF` and by 12× over the standard protocol. Our total communication improvement is lower than our preprocessing improvement because our evaluation phase communication is not improved. While improvement over the standard protocol is almost constant across runs, the improvement over `MOTIF` differs due to varying circuit topology: our improvement ranges from 2.16× to 2.93×.

8.3 Wall-Clock Time

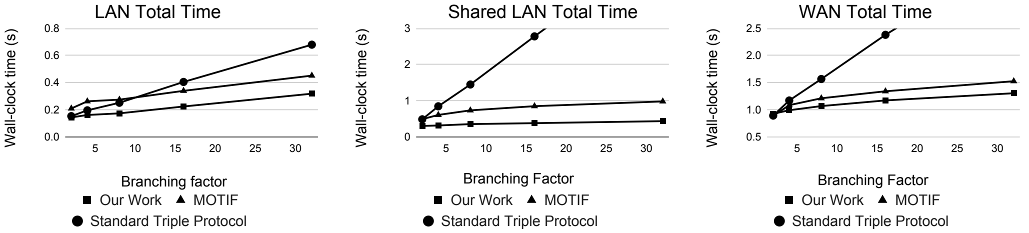

Fig. 11: 2PC comparison of Π_{MT} against `MOTIF` and a standard triple-based protocol. We plot the following metrics as functions of branching factor: wall-clock time on a LAN (left), the wall-clock time on a LAN where other processes share bandwidth (center), and the wall-clock time on a WAN (right).

We next present the wall-clock time improvements over `MOTIF` and the standard triple-based protocol. We consider three simulated network settings:

1. **LAN:** A simulated gigabit ethernet connection with 1Gbps bandwidth and 2ms round-trip latency.
2. **Shared LAN:** A simulated local area network connection where the protocol shares network bandwidth with a number of other processes. The connection features 50Mbps bandwidth and 2ms round-trip latency.
3. **WAN:** A simulated wide area network connection with 100Mbps bandwidth and 20ms round-trip latency.

Figure 11 plots total wall-clock time for each network setting.

Discussion. In these metrics, Π_{MT} is preferred:

- **LAN wall-clock time.** On a fast LAN, our approach's improvement is diminished compared to our communication improvement. On average and for 16 branches, we improve by 1.52× over `MOTIF` and by 1.81× over the standard protocol. A 1Gbps network is very fast, and our modest hardware struggles to keep up with available bandwidth.
- **Shared LAN wall-clock time.** On the more constrained shared LAN, our hardware easily keeps up with the communication channel, and we see corresponding improvement. On average and for 16 branches, we achieve 2.26× speedup over `MOTIF` and 7.43× speedup over the standard protocol.
- **WAN wall-clock time.** On this high-latency network our advantage is less pronounced. On average and for 16 branches, we achieve 1.14× speedup over `MOTIF` and 2.04× speedup over the standard protocol. This high-latency network highlights the weakness of multi-round protocols in such settings.

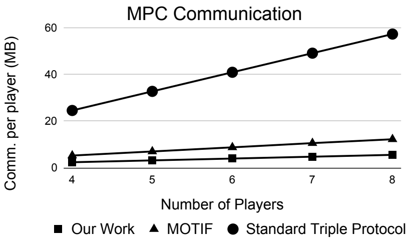

Fig. 12: Protocol per-party communication usage as a function of the number of parties. Like `MOTIF` and the standard protocol, we consume per-party communication linear in the number of parties.

8.4 Scaling to MPC

Our last experiment emphasizes our approach's scaling to the multiparty setting. This experiment uses the same circuit parameters as the former experiments, but we fix the number of branches to 16. We implemented `Π-MaskGen-VS` and ran the circuit among 4-8 parties. Figure 12 plots per-party communication as a function of the number of parties.

Discussion. $\mathit{\Pi}_{\mathsf{MT}}$ works well in the multiparty setting. Our optimization does not add additional costs as compared to `MOTIF` and standard triple-based protocol. Each technique consumes communication quadratic in the number of parties.

Acknowledgements. This work was supported in part by NSF award #1909769, by a Facebook research award, and by Georgia Tech's IISP cybersecurity seed funding (CSF) award.

References

[AGKS19] Masaud Y. Alhassan, Daniel Günther, Ágnes Kiss, and Thomas Schneider. Efficient and scalable universal circuits. Cryptology ePrint Archive, Report 2019/348, 2019. `https://eprint.iacr.org/2019/348`.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[BCG$^+$19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BIB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

[BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 215–232. Springer, Heidelberg, September 2020.

[CDE$^+$18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[GKS17] Daniel Günther, Ágnes Kiss, and Thomas Schneider. More efficient universal circuit constructions. Cryptology ePrint Archive, Report 2017/798, 2017. `http://eprint.iacr.org/2017/798`.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[HK20a] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Heidelberg, August 2020.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[HKP20] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 3–30. Springer, Heidelberg, December 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[Kol18] Vladimir Kolesnikov. `Free IF`: How to omit inactive branches and implement S-universal garbled circuit (almost) for free. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 34–58. Springer, Heidelberg, December 2018.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In Gene Tsudik, editor, *FC 2008*, volume 5143 of *LNCS*, pages 83–97. Springer, Heidelberg, January 2008.

[KS16] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 699–728. Springer, Heidelberg, May 2016.

[LMS16] Helger Lipmaa, Payman Mohassel, and Saeed Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017, 2016. `http://eprint.iacr.org/2016/017`.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, Heidelberg, August 2014.

[LYZ$^+$20] Hanlin Liu, Yu Yu, Shuoyao Zhao, Jiang Zhang, and Wenling Liu. Pushing the limits of valiant's universal circuits: Simpler, tighter and more compact. *IACR Cryptology ePrint Archive*, 2020:161, 2020.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NW78] Albert Nijenhuis and Herber S. Wilf. *Combinatorial Algorithms For Computers and Calculators.* AP Academic Press, New York, NY, USA, 1978.

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *STOC*, pages 196–203, New York, NY, USA, 1976. ACM Press.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[ZYZL18] Shuoyao Zhao, Yu Yu, Jiang Zhang, and Hanlin Liu. Valiant's universal circuits revisited: an overall improvement and a lower bound. Cryptology ePrint Archive, Report 2018/943, 2018. `https://eprint.iacr.org/2018/943`.