Garbling, Stacked and Staggered
Faster k-out-of-n Garbled Function Evaluation

David Heath, Vladimir Kolesnikov, and Stanislav Peceny

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

Abstract. Stacked Garbling (SGC) is a Garbled Circuit (GC) improvement that efficiently and securely evaluates programs with conditional branching. SGC reduces bandwidth consumption such that communication is proportional to the size of the single longest program execution path, rather than to the size of the entire program. Crucially, the parties expend *increased computational* effort compared to classic GC.

Motivated by procuring a subset in a menu of computational services or tasks, we consider GC evaluation of k-out-of-n branches, whose indices are known (or eventually revealed) to the GC evaluator E. Our stack-and-stagger technique amortizes GC computation in this setting. We retain the communication advantage of SGC, while *significantly* improving computation *and wall-clock time*. Namely, each GC party garbles (or evaluates) the total of n branches, a significant improvement over the $O(n \cdot k)$ garblings/evaluations needed by standard SGC. We present our construction as a garbling scheme.

Our technique brings significant overall performance improvement in various settings, including those typically considered in the literature: e.g. on a 1Gbps LAN we evaluate 16-out-of-128 functions $\approx 7.68\times$ faster than standard stacked garbling.

Keywords: Garbled Circuit, Conditional Branching, Stacked Garbling

1 Introduction

Garbled circuits (GCs) allow two mutually untrusting parties to securely compute arbitrary functions of their private inputs while revealing only the functions' outputs. GC was formalized by [BHR12] as a primitive. It is foundational in secure multiparty computation (MPC).

Research that improves the practical cost of GC, e.g. [NPS99, KS08, BHKR13, KMR14, ZRE15, BMR16], focuses on two metrics:

- *Communication.* GC constructions require that the GC generator G send to the GC evaluator E a large collection of ciphertexts that encode the truth tables for each gate. Reducing the number of required ciphertexts improves network utilization.

- *Computation.* Typical GC constructions encode each gate's truth table using a hash function or key derivation function H. Reducing the number of calls to H improves each party's CPU utilization.

Stacked Garbling (or SGC) [HK20c, HK20b] is a recent GC improvement that reduces GC communication consumption for functions that contain *conditional branching*, e.g. as the result of a program if or switch statement. SGC shows that G need not send material proportional to the entire circuit; sending material proportional only to the longest program execution path is sufficient.

While traditionally communication is considered the bottleneck in the GC performance, SGC's communication improvement changed that *status quo*, as it *did not* bring a corresponding computation improvement. In many settings, computation now limits GC performance, sometimes severely (see examples and use cases discussed in our work).

SGC improves evaluation of 1-out-of-n circuits. In this work, we consider a generalization to secure evaluation of k-out-of-n circuits, but with the constraint that the GC evaluator E knows (or learns) the identities of the active branches. Such evaluation is well motivated, see Section 1.1. While SGC can be directly employed to solve this problem, the resulting solution is computationally inefficient: we could use the 1-out-of-n SGC solution k times, but this leads to each party (i.e. G and E) garbling each of n branches $O(k)$ times[1]. Already for $k > 3$ computation may overtake communication as the limiting resource (see, e.g., Figure 6).

This overhead is unfortunate, particularly because standard garbling without stacking does not require this computation overhead. In standard garbling, the generator G simply generates each of the n circuits once and includes a special multiplexer circuit that propagates the output of the k active branches only. Thus each party processes a circuit only $O(n)$ times. However, adopting this approach sacrifices the communication benefit of SGC.

Thus, if we wish to securely evaluate k-out-of-n branches, we must either compromise computation or communication. Fortunately, this dilemma can be resolved, and we can get the best of both above techniques.

In this work, we show that we can retain the communication advantage of SGC, while only garbling an (almost) optimal number of branches. Specifically, like the standard SGC-based approach, we consume communication proportional to only k circuits, while reducing the computation to n garblings by G and $n-k$ garblings by E(compare to the total of SGC's $O(n \cdot k)$ garblings). The resulting wall-clock time improvement is surprisingly significant, with the total runtime almost $k\times$ smaller than SGC for a wide range of parameters (see Section 10).

In sum, we note that network bandwidth is a limited resource, and should be consumed with care. At the same time, it is not pragmatic to reduce communication by indiscriminately sacrificing computation: even mainstream CPUs

[1] Because of our setting, we consider a corresponding special case of SGC where E knows the active branch; hence to evaluate 1-out-of-n functions, the parties consume only linear work in n. General SGC, where neither party knows the active branch, requires $O(n \log n)$ computation [HK21].

generate GC at the rate only 3× the speed of 1Gbps LAN, and are expensive both in dollar cost and power consumption. Our approach strikes a balance between network utilization and hardware utilization and potentially allows use with cheap, computationally weak devices whose use, e.g. in IoT, is exploding.

Our Setting. We summarize our considered setting.

GC generator G and evaluator E agree on a set of n functions of which k will be evaluated. Both G and E provide input into the functions (common input can be reused across functions).

E knows *a priori* or receives as output from the GC the identity of the k branches. If the active branches are revealed by the GC, then the revelation must be completed before the k-out-of-n conditional can run.

We formalize our approach as a *garbling scheme* [BHR12], not a protocol. Garbling schemes are flexible primitives that can be plugged into a number of cryptographic protocols. For example, our implementation (Section 10) uses our scheme to implement a typical constant-round 2PC protocol secure against semi-honest adversaries.

The k-out-of-n branching can be sequentially composed and nested (see discussion in Sections 7.3 and 7.4).

1.1 Motivation

Consider a server that offers a suite of various services to clients, and suppose that these services have privacy concerns for both the server and client. The client may *a priori* know which services it wishes to request, but may wish that even the choice of services are kept secret. Alternatively, the identity of the provided services might be computed securely, but might be implied by the client's output. In such cases, an efficient k-out-of-n secure computation can allow the server to securely provide k out of its offered n services while learning only the number of requested services k.

As an example, suppose a telehealth company offers services that screen concerned patients for a variety of medical conditions. In this example, both parties may have privacy concerns: the patient may not wish to disclose her health data and the server may use sensitive health data of other individuals to aid in the screening or use proprietary data. The patient may *a priori* know that a number of medical conditions are unlikely to be the source of her symptoms (e.g. it is unlikely that the patient's headaches are caused by athlete's foot). Hence, the client may only wish to be screened for k health conditions out of the possible n. By employing our k-out-of-n construction, the telehealth company can potentially offer its services to the client at a cheaper rate.

We note that high speed networking (e.g., LAN or WiFi modules) are cheap (a few U.S. dollars) and low-power, while even mainstream CPUs are expensive both in cost and power. Thus, our approach is particularly suited for use with cheaper computationally weaker and/or battery-powered devices, whose use, e.g., in IoT, is exploding.

1.2 Contribution

We improve GC evaluation of k-out-of-n functions where the k choices are known (either *a priori* or revealed by the computation) to the GC evaluator E. In particular we:

- Present a modification to Stacked Garbling [HK20b] that retains $O(k)$ communication consumption, but that improves the total number of branch garblings performed by G and E from $O(n \cdot k)$ to only $2n - k$.
- Prove our construction secure as a *garbling scheme* [BHR12]. Garbling schemes can be used as a primitive to instantiate secure protocols, e.g. semi-honest 2PC protocols.
- Implement and experimentally evaluate our approach. The implementation instantiates a semi-honest 2PC protocol. Our experimental results indicate that our computation improvement (and wall clock time!) over standard Stacked Garbling indeed scales with k. For example, for $n = 128$ and $k = 16$ we improve over Stacked Garbling by $\approx 7.68\times$.

1.3 High Level Intuition

Stacked Garbling improves communication needed to evaluate 1-out-of-n circuits by bitwise XORing, or *stacking*, the n GC *materials* needed to evaluate each of the branches. For each inactive branch, E receives a compact seed used to derive all of G's randomness when garbling it. This allows E to securely reconstruct and unstack material for the $n - 1$ inactive branches, so that she can recover the material for the active branch and evaluate normally. Unfortunately, naively extending this technique to k-out-of-n branches incurs factor k increase in the cost to garble and unstack material.

At the highest level, our technique shows that G can send to E k stacks of *linearly independent* combinations of exactly n materials. Crucially, each stack will now contain garblings of both inactive and active branches. The inactive garblings can be easily removed from these k stacks via seeds. The remnants are a collection of k stacks that each contain k active materials. To transform these k stacks into the k materials, we use the linear independence: E uses an optimized form of Gaussian elimination to quickly transform the k stacks into k materials and then uses the results to evaluate the k active circuits.

Because we reuse the same n materials across all k branch evaluations, we reduce garbling computation by factor k. We choose our linear combinations carefully such that this stacking and unstacking can be achieved using only simple XOR operations. Our technique must consume $O(n \cdot k)$ calls to XOR to stack/unstack the linear combinations, but these XOR calls are significantly cheaper than the gate-by-gate construction of circuit garblings. Hence our technique significantly improves performance.

2 Related Work

Stacked Garbling. Ours is in a line of works that improve GC evaluation of conditional branches [KKW17, Kol18, HK20c, HK20b, HK21]. The more recent

of these works introduce Stacked Garbling, improving GC communication by up to the program branching factor.

While [HK20b] and [HK21] consider the general case of branching where neither party knows the active branch, [Kol18] and [HK20c] consider special cases where one party knows the branch. Our work builds on [HK20c] which considers the case where E knows the active branch, and we review its technique in Section 4.

[Kol18] considers the dual case where G knows the active branch. We briefly explain why we do not instead build on this work.

Firstly, the setting is less flexible, since G, unlike E, cannot non-interactively receive messages from the GC. Hence G must *choose* the active branches. In contrast, our approach allows E to choose the active branches *or alternatively* allows the branches to be chosen by the GC and revealed to E as part of her output.

Secondly, GC players G and E enjoy asymmetric levels of trust: security against malicious E is often trivially attained simply from the authenticity property of GC. Therefore, changing the roles of the MPC participants (e.g. switching Alice from being G to E) results in a corresponding trust model change, which may not be desired. In the context of the motivation discussed in Section 1.1, G is more naturally played by the (more trusted) service provider.

Finally, and most importantly, the [Kol18] technique also incurs $O(n \cdot k)$ computation, though for different reasons than [HK20c]. In particular, the technique requires G to simply send to E k GC materials corresponding to the active branches. Upon receiving these, E who does not know the branches, tries to evaluate each branch with each material; hence she evaluates $O(n \cdot k)$ times. When E makes a bad guess, her evaluation results in so-called garbage output labels, which must be discarded in favor of valid output labels. [HK20b] showed that garbage can be collected without additional interaction via a special multiplexer, but constructing the multiplexer requires G to emulate E's bad evaluation. Thus, in this construction, both G and E perform $O(n \cdot k)$ times, and it is not clear how this can be improved. In contrast, the [HK20c] technique allows for an efficient Gaussian elimination technique that we present in this work.

Other Improved Secure Conditional Branching. Outside GC, improved conditional branching has begun to emerge. Although our emphasis is constant round 2PC, we mention these works for completeness.

In the Zero Knowledge setting, some branching improvements have been made. [GGHAK21] developed a compiler for sigma protocols that takes advantage of disjunctive proofs. [BMRS20] developed an efficient interactive ZK proof system that incorporates a stacking optimization. Like our work, [BMRS20] also considers k-out-of-n branching, though their protocol and setting are entirely different.

In the MPC setting, conditional improvements to the classic GMW protocol and to Beaver Triples have been shown [HKP20, HKP21].

Prior work motivated by similar scenarios. We discuss several prior works, whose setting and motivation is related to ours. These works address different MPC aspects, and we don't compare our work to them, e.g., w.r.t. performance.

Multiple executions of (identical) functions was considered by [HKK+14, LR14]. Both works designed improved cut-and-choose algorithms for batched execution. We can view our work as improving a batched execution of *a subset of different* functions. We note that improvements of bare garbling schemes are more rare than improvements in the richer world of constructions built on garbling schemes.

[KNT06] considered private policy negotiation. Here the negotiation process itself, i.e. determining what data will be revealed under what conditions, is considered privacy-sensitive. One method of policy selection considered in this work involves one player privately evaluating k-out-of-n matching functions. A related question is of multi-factor authentication or policy match check, where authentication (policy check) succeeds if k out of n private conditions hold.

3 Notation and Assumptions

Notation.

- G is the GC generator. We refer to G as he/him.
- E is the GC evaluator. We refer to E as she/her.
- κ denotes the computational security parameter (e.g. 128).
- $[n]$ denotes the sequence of natural numbers $0, ..., n-1$.
- We work with vectors/matrices and bitstrings (i.e., vectors of bits). We index vectors and matrices with subscripts and use 0-based indexing, e.g. x_0 or $A_{i,j}$.
- We consider GC evaluation of k-out-of-n circuits:
  - n is the number of branches.
  - k is the number of *target* branches.
  - $\mathcal{C}_i$ is the Boolean circuit that implements branch i.
  - M denotes the vector of n *materials* corresponding to the garbling of each branch. Informally, the material M_i is the collection of encrypted truth tables needed for E to evaluate branch $\mathcal{C}_i$.
  - t denotes the *target set*, which is the set of active branches. Since t always has size k, we treat it interchangeably as a vector of k elements.

Cryptographic Assumptions. Our garbling scheme (Section 7) requires only standard assumptions.

Our *implementation* builds on top of the state-of-the-art Boolean circuit half-gates technique [ZRE15] which uses the Free XOR technique [KS08]. Thus our implementation assumes a circular correlation robust hash function H [CKKZ12].

4 Preliminaries – Stacked Garbling [HK20c]

Our work can be viewed as an extension to the Stacked Garbling (SGC) technique [HK20b, HK20c] which improves the communication consumption of GC in the context of conditional branching.

Even though our focus is secure 2PC, not ZK, the most relevant to our work is the technique of [HK20c], which improves GC-based ZK (GCZK). Indeed, their core technique does not actually *require* the simpler ZK setting. Instead, it simply requires that the GC evaluator E knows the identity of each active branch. In such cases, [HK20c] shows that it suffices to send garbled material proportional to the longest conditional branch rather than to the entire circuit.

[HK20c] is built on two ideas:

1. The material produced by garbling each conditional can be handled as a bitstring. This means that materials can be XORed, or *stacked*, with one another to reduce communication.
2. Material can be expanded from a pseudorandom seed. If all random choices used to generate a circuit garbling are derived from a seed, then material is a deterministic expansion of that seed. Thus, a seed is a compact representation of a circuit material.
   As a seed uniquely determines all wire labels in the GC, it is insecure to send material via a seed to the circuit evaluator. However, [HK20c] shows that it is secure to reveal a seed for an *inactive* branch.

Let $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ be a set of conditionally composed circuits. Let $\mathcal{C}_t$ be the active (target) branch and let E know t. G knows each $\mathcal{C}_i$, but does not know and must not learn t.

G chooses n seeds s_i and uses each s_i to garble circuit $\mathcal{C}_i$. The result is a vector of n 'materials' M, which are the collections of encrypted truth tables needed to evaluate the GC. Rather than sending the concatenation of these materials, G instead *stacks* the materials by sending to E the following XOR sum:

$$\bigoplus\nolimits_i M_i$$

(Shorter materials are padded with trailing zeros such that each material has the same length.) Of course, the XOR sum of the n materials is shorter than the concatenation, and hence SGC greatly reduces communication consumption.

The parties then ensure that E receives the $n-1$ seeds $s_{i\neq t}$ corresponding to each inactive branch. These seeds can be conveyed from G to E via oblivious transfer (OT) or can be conveyed by the GC itself [HK20b][2]. Because E holds these seeds, she can reconstruct each material $M_{i\neq t}$ by simply replaying the actions of G. E now computes:

$$\left(\bigoplus\nolimits_i M_i\right) \oplus \left(\bigoplus\nolimits_{i\neq t} M_i\right) = M_t$$

[2] Namely, at runtime E will hold garbled labels corresponding to the branch condition t; G can include an encrypted table that allows E to decrypt different seeds depending on the semantic value of t. Thus, E can receive the seeds without additional interaction. This is useful when t is implied by E's output rather than by her input.

Therefore E can recover the material M_t corresponding to the active branch. Crucially, although E successfully recovers the branch material, she never receives the sensitive seed s_t[3]. From here, E can evaluate the active branch normally. We can exit the conditional and continue GC evaluation by including a multiplexer component[4]. Thus, [HK20c] evaluates 1-out-of-n circuits while requiring only enough communication for the single longest circuit.

5 Technical Overview

Let $\mathcal{C}_0, \ldots, \mathcal{C}_{n-1}$ denote n circuits of which k should be evaluated. For sake of example, suppose that $k = 2$; our approach generalizes to arbitrary k. Let α, β denote the indices of the two circuits to be evaluated.

Let us first consider evaluation of circuit $\mathcal{C}_\alpha$. As reviewed in Section 4, standard Stacked Garbling [HK20c] allows the GC generator G to first generate each circuit $\mathcal{C}_i$ from a seed s_i. Let M denote the vector of n resulting GC materials (i.e., the collections of encoded gate truth tables). Rather than sending the concatenation of these materials, G instead sends to E the following XOR sum:

$$\bigoplus\nolimits_i M_i$$

E then receives each seed $s_{i\neq\alpha}$, uses these seeds to regenerate each material $M_{i\neq\alpha}$, and then computes:

$$\left(\bigoplus\nolimits_i M_i\right) \oplus \left(\bigoplus\nolimits_{i\neq\alpha} M_i\right) = M_\alpha$$

Thus, E can recover the material for the target branch using communication proportional to only the single longest material. From here, E can use GC input labels to correctly evaluate circuit $\mathcal{C}_\alpha$.

Now, consider evaluation of $\mathcal{C}_\beta$. Unfortunately, the above work cannot be re-used when evaluating $\mathcal{C}_\beta$. Namely, it is *not secure* for G to re-use any above materials M_i: E has already received s_β, and it is not secure for E to hold a seed used to generate evaluated material. Instead, G must start from fresh seeds s'_i and generate fresh materials M'. Similarly, E must receive all seeds $s'_{i\neq\beta}$ and regenerate each material $M'_{i\neq\beta}$. In general, E and G each generate each of n circuits k times.

[3] The seed s_t conceptually contains both the material M_t *and* all 'wire labels', and so it is not secure for E to view this seed. If she did, she could decrypt the wires on the active branch. It *is* secure for her to receive seeds on inactive branches because we can ensure that inactive branches hold no semantic values via a *demultiplexer* component [HK20b].

[4] [HK20b] and [HK21], which consider the more general case where neither party knows the active branch, require significant extra computation to generate the multiplexer. This extra computation is needed because E does not know the active branch t, and hence makes "mistakes" during evaluation that need to be cleaned up. [HK20c] and we assume that E does know t, and hence the multiplexer is extremely efficient to generate by simply enumerating a garbled table based on t.

Our Approach. Let us return to the point where G had computed each material M_i, but before any stacking and sending had taken place. We allow the parties to re-use the same n materials M_i to evaluate *all* k target circuits. Thus, we reduce the number of needed materials from $n \cdot k$ to only n.

Our key idea is to view each material M_i as an element in a (very large) finite field $\mathrm{GF}(2^\ell)$. From here, G computes and sends to E k linearly independent combinations of the n materials. For example, when $k = 2$, G sends the following two stacks:

$$M_0 \oplus M_1 \oplus M_2 \oplus \cdots \oplus M_{n-1}$$
$$M_0 \oplus 2 \cdot M_1 \oplus 4 \cdot M_2 \cdots \oplus 2^{n-1} \cdot M_{n-1}$$

The GC then conveys to E (via a garbled gadget) all seeds $s_{i \neq \alpha,\beta}$; hence E can reconstruct all materials $M_{i \neq \alpha,\beta}$. This information suffices for E to perform Gaussian elimination. In particular, E XORs both stacks with (multiples of) her reconstructed materials and hence recovers:

$$M_\alpha \oplus M_\beta$$
$$2^\alpha \cdot M_\alpha \oplus 2^\beta \cdot M_\beta$$

E can now solve for M_α and M_β and then use these materials to securely evaluate $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$. Crucially, (1) we re-use the same n materials to evaluate all k circuits, (2) we retain SGCs $O(k)$ communication complexity, and (3) we ensure that E does not obtain the seed for any active branch. Hence, we can securely and efficiently evaluate k-out-of-n functions inside the GC.

5.1 Gaussian Elimination via 'Staggering'

In practice, we choose our field $\mathrm{GF}(2^\ell)$ and our linear combinations such that all operations can be implemented by simple XORs. This ensures that we can both stack and unstack material using efficient hardware instructions. Hence, the approach is efficient in practice.

Specifically, the materials in each stack are 'staggered' (see Figure 1) by multiplying them by some power of two in the field. The powers of two for each stack are chosen such that all stacks are linearly independent combinations of the materials and hence contain sufficient information to unstack. We choose the field $\mathrm{GF}(2^\ell)$ such that even multiplying a material by the largest such power of two will not cause "wrap-around" in the field, and hence no modular reduction is needed to implement multiplication. Formally, if m is the size of the largest material, $\ell \geq m + n \cdot k$ is (more than) sufficient to ensure no modular reduction is needed. From here on, we largely ignore the size of the field and simply assume it is large enough to avoid need for modular reduction.

6 Garbling, Stacked and Staggered

We now formalize the key algorithms for our staggered stacking technique discussed above. Section 7 later hosts these algorithms in a *garbling scheme* [BHR12] such that our technique can be used in GC protocols.

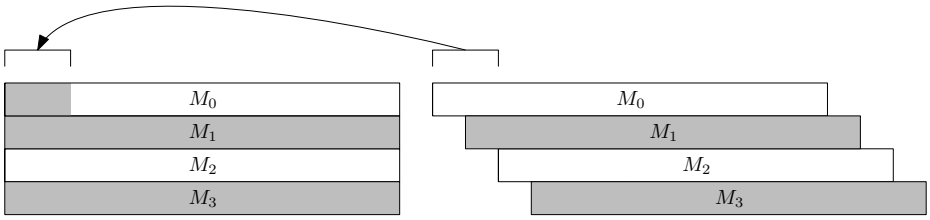

Fig. 1: An example of our staggered stacking for $n = 4$ and $k = 2$. Suppose $\mathcal{C}_0$ and $\mathcal{C}_2$ are the active branches. Hence E reconstructs from seeds inactive branch materials M_1 and M_3. G sends to E two stacks of all material, but the materials in the second stack are "staggered" by prepending materials with varying numbers of zeros. E unstacks (indicated in grey) M_1 and M_3. Notice that E can directly extract the first bits of M_0 from the second stack. She then unstacks these bits from the first stack, allowing her to view the first bits of M_2. By repeatedly unstacking portions of materials from the stacks, E eventually unstacks all k target materials.

PARAMETERS:

- A finite field $\mathbb{F} \triangleq GF(2^\ell)$.
- Number of branches n.
- Number of targets k.
- The stacking matrix $\mathbb{A}$ for n and k (see Definition 1).

INPUT: A vector of materials $M \in \mathbb{F}^n$.
OUTPUT: A vector of stacks of material $S \in \mathbb{F}^k$ where $S \triangleq \mathbb{A} \cdot M$.
PROCEDURE:

$S \leftarrow 0^k$
for $i \in [k]$:
  for $j \in [n]$:
    if $\mathbb{A}_{i,j} \neq 0$:
      ▷ $\ll$ denotes bitwise left shift.
      $S_i \leftarrow S_i \oplus (M_j \ll \log \mathbb{A}_{i,j})$
return S

Fig. 2: G's procedure for stacking n materials into k stacks. The procedure implements simple matrix multiplication by the stacking matrix $\mathbb{A}$ (see Definition 1). We take advantage of zeros in $\mathbb{A}$ to reduce the number of needed XOR operations.

Figure 1 depicts the bit-shift interpretation of our staggering technique for $n = 4$ and $k = 2$. We expound on this intuition, formalizing the technique for arbitrary n and k by using our field multiplication interpretation.

PARAMETERS:

- A finite field $\mathbb{F} \triangleq GF(2^\ell)$.
- Number of branches n.
- Number of targets k.
- Maximum material length $\text{len} < \ell$.
- The stacking matrix $\mathbb{A}$ for n and k (see Definition 1).

INPUT:

- A set of targets t.
- A vector of stacks $S' \in \mathbb{F}^n$ where materials for inactive branches $\mathcal{C}_{i \notin t}$ have been XORed out.

OUTPUT: A vector of materials for the active branches $\mathcal{C}_{i \in t}$. I.e., the vector of materials $M_{i \in t} = \text{Strike}(t, \mathbb{A})^{-1} \cdot S'$.
PROCEDURE:

$$\text{delay}_i \triangleq \begin{cases} t_0 \cdot (k-1) & \text{if } i = k-1 \\ \text{delay}_{i+1} + i \cdot (t_{k-j-1} - t_{k-j-2}) & \text{otherwise} \end{cases}$$

$\text{out} \leftarrow 0^k$

▷ Iterate through the stacks bit by bit.

for $b \in [\text{len} + \text{delay}_0]$:

  ▷ Iterate through each stack, recovering and unstacking one bit per stack.

  for $i \in k-1, ..., 0$:

    if $\text{delay}_i \leq b < \text{delay}_i + \text{len}$:

      ▷ α denotes the branch index *associated* with stack S_i.

      $\alpha \leftarrow t_{k-i-1}$

      $p \leftarrow b - \text{delay}_i$

      ▷ shift denotes the bit index of the bth bit of M_α.

      $\text{shift} \leftarrow \log(A_{i,\alpha}) + p$

      $\text{out}_{i,p} \leftarrow S'_{i,\text{shift}}$

      ▷ Unstack the bit $\text{out}_{i,p}$ from each stack.

      for $i' \in [k]$:

        if $\mathbb{A}_{i',\alpha} > 0$:

          $\text{shift}' \leftarrow \log(\mathbb{A}_{i',\alpha}) + p$

          $S'_{i',\text{shift}'} \leftarrow S'_{i',\text{shift}'} \oplus \text{out}_{i,p}$

return out

Fig. 3: Our unstacking algorithm allows E to perform efficient Gaussian elimination and to unstack k materials from k stacks. Strike is defined in Definition 2.

The key object in our formalization is a $k \times n$ matrix which we refer to as the *stacking matrix*. The stacking matrix formalizes the bit shift distance for each material in each stack.

Definition 1 (Stacking Matrix). *Let n be a number of branches and k be a number of targets. The* stacking matrix *for n and k is a $k \times n$ matrix $\mathbb{A}$ whose entries are defined as follows:*

$$\mathbb{A}_{i,j} \triangleq \begin{cases} 0 & \text{if } ((i+j) \bmod n) < k-1 \\ 2^{i \cdot (i+j+k-1)} & \text{otherwise} \end{cases}$$

The stacking matrix is notationally complex, but its structure can be understood through example:

Example 1 (4×6 stacking matrix). The 4×6 stacking matrix is defined as follows:

$$\mathbb{A} \triangleq \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 4 & 0 \\ 0 & 1 & 4 & 16 & 0 & 0 \\ 1 & 8 & 64 & 0 & 0 & 0 \end{bmatrix}$$

Notice that each matrix row has $k-1$ zeros; other entries are powers of two, and the powers of two increase with the rows. We choose varying rows to ensure linear independence which will be important when allowing E to unstack material.

To stack n materials in M into k stacks S, G simply computes $S \triangleq \mathbb{A} \cdot M$. We choose $\mathbb{A}$ with this matrix-vector multiplication in mind. In particular, (1) we maximize the number of zero entries in the matrix and (2) non-zero entries are powers of two. These properties allow for an efficient algorithm that stacks material using only simple XORs (see Figure 2).

G then sends the k stacks S to E and E obliviously receives $n-k$ seeds corresponding to the inactive branches. Let t denote the set of active branch identifiers. From here, E reconstructs the $n-k$ inactive materials: she reconstructs each $M_{i \notin t}$. E then constructs the following n element vector M':

$$M' \triangleq \begin{cases} M_i & \text{if } i \notin t \\ 0 & \text{otherwise} \end{cases}$$

Now, E can remove the $n-k$ reconstructed materials from her stacks by computing the following:

$$S' \triangleq S \oplus \mathbb{A} \cdot M'$$

In other words, E shifts each material by the appropriate amount (according to $\mathbb{A}$) before XORing it with each stack. The resulting vector S' contains linear combinations of the k materials $M_{i \in t}$ only.

To define how E transforms these k stacks into the k active branch materials, we first give a helper definition that allows E to remove columns from the stacking matrix $\mathbb{A}$:

Definition 2 (Strike). *Let A be a $k \times n$ matrix and let s be a set of indices such that $|s| < n$. Then* $\text{Strike}(s, A)$ *is the $k \times (n - |s|)$ matrix that contains all columns in A except those whose index appears in s.*

Example 2 (Strike two columns from 4×6 stacking matrix). Let $\mathbb{A}$ denote the 4×6 stacking matrix and let $s = \{1, 4\}$:

$$\text{Strike}(s, \mathbb{A}) = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 16 & 0 \\ 1 & 64 & 0 & 0 \end{bmatrix}$$

Note that the following equality holds:

$$S' = \text{Strike}(\bar{t}, \mathbb{A}) \cdot M_{i \in t}$$

This equality holds simply from the fact that E already removed each inactive branch material from S'.

Thus, to transform the k stacks into the k target materials, E needs to invert $\text{Strike}(\bar{t}, \mathbb{A})$. The stacking matrix $\mathbb{A}$ is designed with this inversion in mind: by construction, any choice of k columns are linearly independent. Thus, our approach allows E to perform Gaussian elimination to unstack any choice of k-out-of-n branches. To complete the unstacking, E computes the following:

$$M_{i \in t} = \text{Strike}(\bar{t}, \mathbb{A})^{-1} \cdot S'$$

Like our stacking procedure (Figure 2), this unstacking can also be achieved using simple XORs only; see Figure 3. Unlike our stacking procedure, this unstacking procedure is nontrivial. This may be surprising, since the case for $k = 2$ (e.g. Figure 1) is quite simple. However, with $k > 2$ the algorithm must carefully coordinate the order in which bits are unstacked. Nevertheless, due to the special case of the matrix $\mathbb{A}$, we can unstack materials using only $O(k^2 \cdot \ell)$ bit XORs, where ℓ is the length of the longest stack.

We explain in greater detail our unstacking algorithm in the following correctness lemma:

Lemma 1 (Unstack Correct). *Upon inputs t and S', the procedure in Figure 3 indeed computes:*

$$\text{Strike}(\bar{t}, \mathbb{A})^{-1} \cdot S'$$

Proof. By inspection of Figure 3.

Recall, e.g., from Example 1 that each material is shifted by some amount, according to the stacking matrix $\mathbb{A}$. Our algorithm extracts bits one by one from the stacks. Each stack will be used to recover a single material. In particular, stack i will be used to recover material $M_{t_{k-i-1}}$ We say that stack i is *associated* with material $M_{t_{k-i-1}}$ and with branch $\mathcal{C}_{t_{k-i-1}}$.

In the general case, we extract one bit per stack per iteration. However, note that on the first iteration (i.e. before anything has been unstacked), some stacks

may have more than one material stacked into the first bit (for reference, see Figure 1). Therefore, we add a per-stack *delay* which ensures that we do not attempt to recover bits prematurely. For example, in Figure 1, we must first unstack a bit of material from the second stack before we can recover a bit of material from the first.

We formalize the delay for stack i with the following expression:

$$\text{delay}_i \triangleq \begin{cases} t_0 \cdot (k-1) & \text{if } i = k-1 \\ \text{delay}_{i+1} + i \cdot (t_{k-j-1} - t_{k-j-2}) & \text{otherwise} \end{cases}$$

This delay formalizes the distance between (1) the bit location in the ith stack where its associated material begins and (2) the bit location in the $(i+1)$th stack where its associated material begins. Note that the delay is quite complex because we must account for the fact that some materials (i.e. those for inactive branches) have been *a priori* removed from each stack. The delay for stack i depends on (1) the delay of its neighboring stack $i+1$, (2) the shift distance for stack i which is itself i, and (3) the difference between the indexes of the two target branches associated with stack i and stack $i+1$. Note that the smallest delay occurs for stack $k-1$, which corresponds to the fact that we can start extracting material from this stack first.

Now, view the collection of stacks as a matrix S where $S_{i,j}$ denotes the jth bit of stack i. Consider an arbitrary bit $S_{i,j}$. Before we begin to unstack, this bit is a stacking of (in the general case) k bits from each of the k materials. Therefore, we must ensure that $k-1$ bits from the non-associated materials are each removed before we attempt to extract the bit. This is exactly the purpose of the above delay.

We show that our algorithm ensures bit $S_{i,j}$ is correctly unstacked by considering two cases of the other bits originally in $S_{i,j}$: those associated with stack $r > i$ and those associated with stack $r < i$:

1. Consider a bit b associated with stack $r > i$ originally XORed into $S_{i,j}$. By the time our algorithm attempts to extract $S_{i,j}$, b will have already been unstacked. This is ensured because $\text{delay}_r < \text{delay}_i$ by definition.
2. Consider a bit b associated with stack $r < i$ originally XORed into $S_{i,j}$. By the time our algorithm attempts to extract $S_{i,j}$, b will have already been unstacked. This is ensured because we choose delay according to the shift distance of each stack. Thus, although stack r has longer delay than stack i, its associated material is not shifted as much, and hence it will have already been unstacked.

Thus, we correctly recover each bit $S_{i,j}$, and so Figure 3 is correct.

□

Thus G conveys k circuit materials at the cost of only $O(n)$ circuit garblings.

7 Stacked and Staggered Garbling Scheme

We formalize our technique as a *garbling scheme* [BHR12] which we call S&S. S&S allows G and E to securely evaluate k-out-of-n functions where E outputs the identity of the evaluated functions.

A garbling scheme is a five-tuple of algorithms:

$$(\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$$

These five algorithms specify how G and E handle the GC. Namely, (1) En specifies how G encodes cleartext inputs as input labels, (2) Gb specifies how G constructs the garbled circuit, (3) Ev specifies how E evaluates the garbled circuit and obtains output labels, (4) De specifies how output labels are decoded to cleartext outputs, and (5) ev specifies circuit semantics. En, Gb, Ev, and De should together securely implement ev.

Additionally, [BHR12] formalizes the garbling scheme properties *correctness*, *obliviousness*, *privacy*, and *authenticity* (we provide definitions of these properties as we prove them). By proving our garbling scheme satisfies these properties, we ensure that the scheme may be plugged into GC protocols as a black box.

Underlying garbling scheme. Because our focus is conditional evaluation only, we adopt a formalization technique of [HK20c] whereby our scheme focuses *exclusively* on the handling of k-out-of-n branching; we leave the handling of the functions in each branch to another *underlying* garbling scheme which we refer to as Base. S&S may be instantiated with different schemes for Base. Our implementation instantiates Base with the state-of-the-art Boolean-circuit-based half-gates technique [ZRE15].

We require that Base satisfies the [BHR12] properties of *correctness*, *obliviousness*, *privacy*, and *authenticity*. Additionally, [HK21] introduced a property called *strong stackability* which ensures that the garbling scheme produces garbled material that may be safely stacked. We provide their definition of strong stackability; a candidate garbling scheme must satisfy the property to instantiate Base.

Definition 3 (Strong Stackability). *A garbling scheme is* strongly stackable *if:*

1. *For all circuits $\mathcal{C}$ and all inputs x,*
$$(\mathcal{C}, M, \mathsf{En}(e, x)) \stackrel{c}{=} (\mathcal{C}, M', X')$$
where $(M, e, \cdot) = \mathsf{Gb}(1^\kappa, \mathcal{C})$, $X' \leftarrow \{0,1\}^{|X|}$, and $M' \leftarrow \{0,1\}^{|M|}$.
2. *The scheme is* projective *[BHR12]. I.e., the input encoding string e and output decoding string d are vectors of pairs of labels.*
3. *There exists an efficient deterministic procedure* Color *that maps strings to $\{0,1\}$ such that for all $\mathcal{C}$ and all projective label pairs $A^0, A^1 \in d$:*
$$\mathrm{Color}(A^0) \neq \mathrm{Color}(A^1)$$
where $(\cdot, \cdot, d) = \mathsf{Gb}(1^\kappa, \mathcal{C})$.

4. *There exists an efficient deterministic procedure* Key *that maps strings to* $\{0,1\}^\kappa$ *such that for all* $\mathcal{C}$ *and all projective label pairs* $A^0, A^1 \in d$*:*

$$\text{Key}(A^0) \,||\, \text{Key}(A^1) \stackrel{c}{=} \{0,1\}^{2\kappa}$$

where $(\cdot,\cdot,d) = \mathsf{Gb}(1^\kappa, \mathcal{C})$.

The first sub-property ensures that GC material looks random, which ensures that E cannot determine the active branch by simply looking at the garbling[5]. Second, properties (2–4) allow our scheme to work with the output labels that emerge when evaluating Base. More precisely, (3) corresponds to the classic point-and-permute technique to reduce the number of PRF calls in evaluating the GC gates. The Color procedure produces a bit that instructs which garbled row to decrypt. (4) allows us to extract a suitable PRF key from each label.

7.1 Garbling Scheme Algorithms

Construction 1 (S&S Garbling Scheme)**.** *Let* Base *denote an underlying garbling scheme that is* correct*,* oblivious*,* private*,* authentic*, and* strongly stackable*.* S&S *is the tuple of algorithms specified in Figure 4.*

In Section 7.5, we prove Construction 1 *correct*, *oblivious*, *private*, and *authentic*, as defined by [BHR12]. Note that Construction 1 is not strongly stackable; see Section 7.4 for discussion.

The algorithms in Figure 4 host our staggering technique (Section 6) into a garbling scheme. This hosting is relatively straightforward; we note the more interesting details:

- S&S is a *projective* garbling scheme [BHR12]: i.e., each circuit wire has exactly two possible GC labels. Formally, the input encoding string e and output decoding string d are simple vectors of pairs of GC labels. This means that our procedures En and De are standard, and are implemented as straightforward mappings between cleartext values and garbled labels (i.e., they index e and d).
- S&S handles k-out-of-n branching *only*. We leave the handling of low level details of branch internals to Base. Additionally, our scheme can be hosted inside another garbling scheme in order to sequentially compose multiple k-out-of-n computations (see Section 7.3). This way, our formalization can cleanly focus on our contribution without sacrificing expressivity.
- As written, S&S does not directly support *nested* k-out-of-n computations. We discuss nesting, including explaining how it can be added to our scheme,

[5] It may seem strange that we require GC material look random, given that we reveal the active branch to E. However, we need this strong property to meet the [BHR12] definition of obliviousness. Informally, our scheme ensures that the garbling of a circuit provides no information to E. Only once output decoding tables d are revealed to E does E learn the branch conditions, which allow her to evaluate.

$\mathsf{S\&S.Gb}(1^\kappa, \mathcal{C})$:
$k, \mathcal{C}_0, .., \mathcal{C}_{n-1} \leftarrow \mathcal{C}$
$T \leftarrow (\{0,1\}^\kappa)^n$
for $i \in [n]$ **:**
$M_i, e_i, d_i \leftarrow \mathsf{Base.Gb}[T_i](1^\kappa, \mathcal{C}_i)$
$e', M_{\text{dem}} \leftarrow \text{GbDem}(T, e_0..e_{n-1})$
$e \leftarrow T \mid\mid e'$
$d', M_{\text{mux}} \leftarrow \text{GbMux}(T, d_0..d_{n-1})$
$d \leftarrow T, d'$
$M_{\text{stack}} \leftarrow \text{Stack}(k, M_0, .., M_{n-1})$
$M \leftarrow M_{\text{dem}} \mid\mid M_{\text{stack}} \mid\mid M_{\text{mux}}$
return M, e, d

$\mathsf{S\&S.Ev}(\mathcal{C}, M, X, t)$:
$k, \mathcal{C}_0, .., \mathcal{C}_{n-1} \leftarrow \mathcal{C}$
$M_{\text{dem}}, M_{\text{stack}}, M_{\text{mux}} \leftarrow M$
$T, X' \leftarrow X$
$j \leftarrow 0$
for $i \notin t$ **:**
$M_i, e_i, d_i \leftarrow \mathsf{Base.Gb}[T_j](1^\kappa, \mathcal{C}_i)$
$j \leftarrow j + 1$
$M_{i \in t} \leftarrow \text{Unstack}(t, M_{\text{stack}}, M_{i \notin t})$
$X_0, .., X_{k-1} \leftarrow \text{EvDem}(t, T, M_{\text{dem}}, X')$
for $i \in [k]$ **:**
$Y_i \leftarrow \mathsf{Base.Ev}(\mathcal{C}_{t_i}, M_{t_i}, X_i)$
$Y \leftarrow \text{EvMux}(t, T, M_{\text{mux}}, Y_0, .., Y_{k-1})$
return T, Y

$\mathsf{S\&S.ev}(\mathcal{C}, x)$:
$k, \mathcal{C}_0, .., \mathcal{C}_{n-1} \leftarrow \mathcal{C}$
$t, x' \leftarrow x$
$y \leftarrow$ empty-string
for $i \in [k]$ **:**
$y \leftarrow y \mid\mid \mathsf{Base.ev}\,(\mathcal{C}_{t_i}, x')$
return t, y

$\mathsf{S\&S.En}(e, x)$:
$X \leftarrow$ empty-string
for $i \in [|x|]$ **:**
$X^0, X^1 \leftarrow e_i$
if $x_i = 0$ **then** $X_i \leftarrow X^0$;
else $X_i \leftarrow X^1$;
return X

$\mathsf{S\&S.De}(d, Y)$:
$y \leftarrow$ empty-string
for $i \in [|Y|]$ **:**
$Y^0, Y^1 \leftarrow d_i$
if $Y_i = Y^0$ **then** $y_i \leftarrow 0$;
else if $Y_i = Y^1$ **then**
$y_i \leftarrow 1$
else return $\perp$;
return y

Fig. 4: Our garbling scheme S&S. The procedure Stack refers to the procedure specified in Figure 2. The procedure Unstack refers to the procedure described in Section 6 whereby E first computes $S' \triangleq S \oplus \mathbb{A} \cdot M'$ and then performs Gaussian elimination via Figure 3. Our Ev procedure is nonstandard because we pass the set of target branches t as an extra cleartext argument. This models the fact that E knows the targets. T denotes a bitwise encoding of the set t. If T_i encodes zero, then branch $\mathcal{C}_i$ is not active; otherwise $\mathcal{C}_i$ is active. We use the zero encoding for T_i as a seed to garble branch $\mathcal{C}_i$; $\mathsf{Gb}[T_i]$ denotes configuring the randomness in the procedure Gb according to the seed T_i. GbMux and EvMux respectively garble and evaluate the multiplexer component. GbDem and EvDem respectively garble and evaluate the demultiplexer component.

in greater length in Section 7.4. Note that we *can* nest 1-out-of-n oblivious branching inside our scheme by instantiating Base with an existing stacked garbling scheme [HK21].

- Our scheme invokes procedures Stack and Unstack. These procedures use the algorithms in Figures 2 and 3 to stack and unstack material.
- By convention, the first n bits of input to our circuits encode the target set t. The ith target bit indicates if branch $\mathcal{C}_i$ is a target. If branch $\mathcal{C}_i$ is a target, then E will obtain an encoding of one for the ith target bit. Each target bit's zero encoding is used as the seed to garble branch $\mathcal{C}_i$. Hence, E can garble inactive branches.
- We add an additional input t to $\mathsf{S\&S.Ev}$ to model the fact that E knows the target set t. The set t is also formalized as a circuit output. This, again, models the fact that E knows the target set. In particular, this output allows us to prove privacy (see Theorem 3).
- Our top level circuit feeds input to (resp. collects output from) the active branches via a demultiplexer (resp. multiplexer). See below for extended discussion of these gadgets.

7.2 Multiplexer and Demultiplexer

Our construction routes inputs to and collects outputs from the k active branches via a *demultiplexer* and a *multiplexer*. These garbled gadgets are simple, and can be built using standard GC techniques: namely, according to the inputs we build an encrypted function table such that E can decrypt only the appropriate outputs. Both gadgets are similar to the gadgets used in [HK20b, HK21].

For simplicity and because they are constructed in a standard way, we do not fully specify garbled algorithms for these components. However, we do specify the cleartext functions that they compute.

The *demultiplexer* takes as input a target set t and an input string x. For each of the n branches $\mathcal{C}_i$, the demultiplexer computes the following simple function:

$$\text{demux}(t, x) \triangleq \begin{cases} x & \text{if } i \in t \\ \perp & \text{otherwise} \end{cases}$$

Namely, the demultiplexer forwards the input x to each active branch and propagates a garbage value to each inactive branch. In the GC, our demultiplexer encodes $\perp$ values by producing a uniform GC label that is distinct yet indistinguishable from the label's in the encoding string e. This ensures that E does not learn the active branch set t until she sees the output decoding string d, which is needed to show that our scheme is oblivious.

The *multiplexer* takes as input a target set t and n output strings y_i. It concatenates and propagates the output from the k active branches in t:

$$\text{mux}(t, y_0, ..., y_{n-1}) \triangleq y_{t_0} \,||\, ... \,||\, y_{t_{k-1}}$$

7.3 Sequential Composition

As mentioned above, our scheme explicitly handles k-out-of-n branching *only*. Nevertheless, our scheme can be used to achieve more general circuits, where a k-out-of-n conditional may occur multiple times within another circuit (here we cover sequential composition; nested composition is discussed in Section 7.4).

To achieve this kind of sequential composition of circuits, i.e. where another circuit appears before and/or after a k-out-of-n conditional, we can host our scheme inside another garbling scheme. This outer scheme can handle the details of threading the outputs from one circuit to inputs of another. Such sequential composition is not hard, and e.g., the formalization of [HK21] can be used with our scheme to achieve sequential composition.

We emphasize that our scheme requires the outer scheme to pass the target set t as a cleartext argument to $\mathsf{S\&S.Ev}$. This can be easily achieved in both the case where t is part of E's input or if t is computed by a prior circuit component and released as output to E. Technically, the latter release to E is achieved by including t as formal output, and into the corresponding decoding table d. Because t must be available to E prior to the evaluation of the conditional, (at least) the corresponding portion of d must be available to E during GC evaluation.

7.4 Nested Branching

We do not prove our garbling scheme strongly stackable [HK21]. Indeed, our garbling scheme will not in general work for embedding inside a stacked conditional where neither party knows the target branch. Indeed, to use our construction, E must know in cleartext the target set t. Therefore, if $\mathsf{S\&S}$ is in an inactive branch, E will obtain a uniformly sampled set t, which may be distinguishable from a real-execution t, which will break SGC properties.

However, it is easy to see that it is secure to nest *our* scheme inside itself, e.g. to allow a tree of k-out-of-n branching: in this case, the above distinguisher does not apply, since E already knows the targets of all outer conditionals.

Unfortunately, the [BHR12] framework is not suitable for proving that nesting our scheme is secure. The problem is that [BHR12] provides no mechanism for revealing *intermediate* circuit values to E. Rather than either losing the modularity of our scheme or performing a complete overhaul to the [BHR12] framework, we instead provide a modular and informal discussion of the changes that would be needed to prove that nesting our scheme is secure.

Our approach to resolving this is the introduction of the notion of *mandatory outputs*, which will play a special role in composition. Namely, each circuit, in addition to having a collection of *regular* outputs, has a second formal collection of *mandatory* outputs. When composing more than one garbling scheme to build up complexity, the outer scheme is required to verbatim forward all mandatory outputs of its inner schemes as its own mandatory outputs. With this change added, we could set each target set t as a mandatory output. This would in

particular ensure that E learns the control flow path through k-out-of-n conditionals as required. Since E learns the full path through the conditionals, she can correctly evaluate. Moreover, E's view is simulatable since the control flow path is implied by the mandatory outputs.

As discussed above (Section 7.3), these mandatory outputs need to be released over time to E. I.e., to begin evaluating a k-out-of-n conditional, E must know the target set t.

As a final detail, to achieve *obliviousness* (see Definition 5) when nesting branches, our multiplexer and demultiplexer must produce material indistinguishable from uniform strings. This ensures that E cannot learn the active set t before learning the GC decoding string d.

7.5 Proofs

S&S satisfies the [BHR12] definitions of *correctness*, *obliviousness*, *privacy*, and *authenticity*. We include definitions and proofs of each of these properties.

Definition 4 (Correctness). *A garbling scheme is* correct *if for all circuits $\mathcal{C}$ and all inputs x:*

$$\mathsf{De}(d, \mathsf{Ev}(\mathcal{C}, M, \mathsf{En}(e, x))) = \mathsf{ev}(\mathcal{C}, x)$$

where $(M, e, d) = \mathsf{Gb}(1^\kappa, \mathcal{C})$.

Correctness requires that the garbling scheme algorithms implement the semantics specified by ev.

Theorem 1. *If* Base *is correct and strongly stackable,* S&S *is correct.*

Proof. By the correctness and strong stackability of Base.

S&S.En and S&S.De are straightforward mappings between cleartext values and GC labels and so are trivially correct. Thus the core task is to show that given valid input encodings X corresponding to input x, Gb and Ev jointly ensure an output encoding Y corresponding to $\mathsf{ev}(\mathcal{C}, x)$.

Recall that our scheme handles k-out-of-n conditionals only. The conditional is preceded by a demultiplexer, which routes inputs to active branches, and followed by a multiplexer, which collects outputs from the active branches (see Section 7.2).

Recall that for active branch set t, the demultiplexer computes for each branch $\mathcal{C}_i$ the following simple function:

$$\text{demux}(t, x) \triangleq \begin{cases} x & \text{if } i \in t \\ \perp & \text{otherwise} \end{cases}$$

I.e., the input x is routed to each active branch. The demultiplexer (formally GbDem and EvDem) is implemented as a standard garbled gadget built from an encrypted function table and is correct.

The core of our approach is the stacking and unstacking of material. Correctness of this step can be inferred from discussion in Sections 5 and 6. In short, Gb stacks the n circuit materials by left multiplying the stacking matrix $\mathbb{A}$ (see Definition 1 and Figure 2). Recall from Figure 4 that each branch material is constructed using a zero label in the encoding T. Ev computes the inactive branch zero labels and hence can reconstruct and unstack all inactive branch materials. From here, Ev performs Gaussian elimination (via Figure 3, see Lemma 1) to extract the active branch materials. Thus, Ev correctly unstacks materials. Base is assumed correct, so invoking Base.Ev on correct material yields correct output labels for each active branch.

Finally, Ev routes the outputs of each active branch to the multiplexer (formally GbMux and EvMux). Importantly, Base is assumed strongly stackable, so the Color procedure is available. This allows us to construct the multiplexer as a standard garbled gadget based on point and permute [BMR90]. This gadget implements the following simple function:

$$\text{mux}(t, y_0, ..., y_{n-1}) \triangleq y_{t_0} \mid\mid ... \mid\mid y_{t_{k-1}}$$

Hence output labels are properly propagated from the active branches.

$\mathsf{S\&S}$ is correct. □

Definition 5 (Obliviousness). *A garbling scheme is* oblivious *if there exists a simulator $\mathcal{S}_{\text{obv}}$ such that for any circuit $\mathcal{C}$ and all inputs x, the following are indistinguishable:*

$$(\mathcal{C}, M, X) \stackrel{c}{=} \mathcal{S}_{\text{obv}}(1^\kappa, \mathcal{C})$$

where $(M, e, \cdot) = \mathsf{Gb}(1^\kappa, \mathcal{C})$ and $X = \mathsf{En}(e, x)$.

Obliviousness ensures that the material M and encoded input labels X reveal no information about the input x or about the output $\mathsf{ev}(\mathcal{C}, x)$.

Theorem 2. *If* Base *is oblivious and strongly stackable, then* $\mathsf{S\&S}$ *is oblivious.*

Proof. By constructing an obliviousness simulator $\mathcal{S}_{\text{obv}}$.

$\mathcal{S}_{\text{obv}}$ simply does the following: (1) run $\mathsf{S\&S.Gb}(1^\kappa, \mathcal{C})$ to generate a fresh garbling (M', e', d'), (2) run $\mathsf{S\&S.En}(e', 0)$ to generate X', and (3) output $(\mathcal{C}, M', X')$. In other words, the simulator simply constructs a fresh garbling and encodes the all zeros string. We claim that this simulation is indistinguishable from real.

For our k-out-of-n setting, the most notable point is that the garbling does not leak the target set t. The target set t is disclosed to E by the decoding string d which is not available to the obliviousness distinguisher, so t must be hidden. As an informal aside, obliviousness can be useful when E may never eventually evaluate a particular GC, e.g. in cut-and-choose. Hiding the target set t from E until it is explicitly revealed by d allows our scheme to satisfy obliviousness.

Our scheme composes three subcomponents: the n branches themselves, the multiplexer, and the demultiplexer. We consider each.

Recall (from Section 7.2) that the demultiplexer forwards valid input labels to the active branches only; inactive branches are instead given labels that encode $\perp$. The demultiplexer is built as a standard GC gadget, and so it hides the semantic values of its inputs. If branch $\mathcal{C}_i$ is active, then the demultiplexer outputs labels in the encoding string e_i. Each of these encoding strings hold uniform values due to strong stackability (Definition 3). If branch $\mathcal{C}_i$ is inactive, then the demultiplexer instead outputs a uniform string that encodes $\perp$. Hence, the demultiplexer supports indistinguishability. Both in the real world and the simulation, the demultiplexer maps input labels to uniform labels for every branch.

The branches themselves support obliviousness because their garbling is uniform due to strong stackability. This is crucial, since it means that a distinguisher cannot attempt to unstack material to determine which of the t branches are active.

Finally, the multiplexer trivially supports obliviousness since it is built as a standard GC gadget, and maps uniform input labels (due to strong stackability) to output labels.

Hence the simulation is indistinguishable and S&S is oblivious. □

Definition 6 (Privacy). *A garbling scheme is* private *if there exists a simulator $\mathcal{S}_{\text{prv}}$ such that for any circuit $\mathcal{C}$ and all inputs x, the following are computationally indistinguishable:*

$$(M, X, d) \stackrel{c}{=} \mathcal{S}_{\text{prv}}(1^\kappa, \mathcal{C}, y),$$

where $(M, e, d) = \mathsf{Gb}(1^\kappa, \mathcal{C})$, $X = \mathsf{En}(e, x)$, and $y = \mathsf{ev}(\mathcal{C}, x)$.

Privacy ensures that E, who is given (M, X, d), learns nothing about the input x except what can be learned from the output y.

Theorem 3. *If* Base *is oblivious and strongly stackable,* S&S *is private.*

Proof. We prove privacy by constructing a simulator $\mathcal{S}_{\text{prv}}$.

By Theorem 2, S&S is oblivious, and hence there exists an obliviousness simulator $\mathcal{S}_{\text{obv}}$. $\mathcal{S}_{\text{prv}}$ first runs $\mathcal{S}_{\text{obv}}(1^\kappa, \mathcal{C})$, resulting in a simulated garbling $(\mathcal{C}, M', X')$. From here, $\mathcal{S}_{\text{prv}}$ must construct a decoding string d' that together with M' and X', is indistinguishable from (M, X, d) even given the output y.

We now show how $\mathcal{S}_{\text{prv}}$ simulates d.

$\mathcal{S}_{\text{prv}}$ holds M' and X'; it also knows the set t of the target branches. (Recall by our syntactic convention, t is always included in the output of S&S.) $\mathcal{S}_{\text{prv}}$ uses these strings to invoke $Y' = \mathsf{Ev}(\mathcal{C}, M', X', t)$ and obtains Y', which holds the output labels for all target branches. The key issue is to now simulate d' such that $\mathsf{S\&S.De}(d', Y') = y$. Indeed, if this decoding does *not* hold, then there is clearly a distinguisher. Constructing such d', given Y' and y is easy: $\mathcal{S}_{\text{prv}}$ simply populates the 2-dimensional table of d' with the corresponding to y labels of Y'. It then fills in the remaining slots of d' with simulated labels. Note, here we rely on the property that unseen labels of the GC can be simulated. This holds for

standard GC schemes, e.g. half-gates, standard Yao, etc. Hence the multiplexer, which ultimately produces our output labels and is built using such standard techniques, produces simulatable labels.

It is easy to see that this simulation is indistinguishable from the real execution. Indeed, the simulated d' successfully decodes the true output y, and its entries are indistinguishable from the entries of the real d. □

Definition 7 (Authenticity). *A garbling scheme is* authentic *if for all circuits $\mathcal{C}$, all inputs x, all target sets t, and all poly-time adversaries $\mathcal{A}$ the following probability is negligible in κ:*

$$\Pr\left(Y' \neq \mathsf{Ev}(\mathcal{C}, M, X, t) \wedge \mathsf{De}(d, Y') \neq \bot\right)$$

where $(M, e, d) = \mathsf{Gb}(1^\kappa, \mathcal{C})$, $X = \mathsf{En}(e, x)$, and $Y' = \mathcal{A}(\mathcal{C}, M, X, t)$.

Authenticity ensures that even an adversarial E cannot construct labels that successfully decode except by running Ev as intended.

Theorem 4. *If* Base *is authentic and strongly stackable,* S&S *is authentic.*

Proof. Authenticity follows from the fact that (1) the multiplexer and the demultiplexer are implemented as garbled gadgets using standard GC that is authentic and (2) Base is assumed authentic.

Our proof starts at the end of a circuit and proceeds backwards, at each step showing that $\mathcal{A}$ cannot forge outputs of a circuit component without forging inputs to that component. Thus, $\mathcal{A}$ cannot forge overall circuit outputs without forging overall circuit inputs, and so the circuit is authentic.

Recall that our scheme handles k-out-of-n circuits only, so we need only prove the related subcomponents authentic.

- As stated above, the multiplexer is built as a standard authentic GC gadget. It is authentic.
- The n branches are authentic by assumption on Base. Note that $n - k$ of these branches are *inactive* and $\mathcal{A}$ holds the seeds for each of these branches. Hence, she may forge arbitrary outputs from each inactive branch. However, the logic of the multiplexer component discards inactive branch outputs, so forging values inside inactive branches does not help $\mathcal{A}$ forge outputs of the overall conditional.
- The demultiplexer, like the multiplexer, is a standard authentic GC gadget.

Note that our scheme can compose the three above components because of the strong stackability of Base: namely, our multiplexer and demultiplexer can directly manipulate the wire labels of the base scheme. Strong stackability ensures this manipulation is authentic (resp. correct) by including Key (resp. Color) procedures.

S&S is authentic. □

8 Application to Zero Knowledge Proofs (ZKP)

We point out an immediate application of our garbling scheme to Zero Knowledge. ZK is a natural application for the setting where the k choices are known to the GC evaluator E (prover in GC-ZK). Our garbling scheme can be directly used in GC-ZK, resulting in a corresponding computation improvement from $O(n \cdot k)$ to only $2n - k$ of symmetric key operations. Unlike the MPC setting, our k-out-of-n branching does not place additional assumptions on the prover; in particular, branching can be placed anywhere in the proof statement. We don't view this as our core contribution, since IT-MAC-based ZK [HK20a, WYKW21, YSWW20, BMRS21, DIO20] overtook GC-ZK in performance in the area of interactive ZK.

9 Implementation and Experimental Setup

We implemented S&S in C++ and used it to instantiate a semi-honest 2PC protocol such that we can evaluate our approach (see Section 10).

Implementation Details. We instantiated the underlying garbling scheme Base with [ZRE15]'s state-of-the-art half-gates technique. Our computational security parameter κ is 127: we reserve the 128th bit for the classic permute-and-point technique [BMR90]. We instantiated oblivious transfer, needed to convey input labels from G to E, via the OT extension of [IKNP03] as implemented by EMP [WMK16].

Our formal constructions present our staggering technique at the bit-level. I.e., the stacking matrix $\mathbb{A}$ (Definition 1) staggers stacks by low powers of two. To achieve higher concrete efficiency, our implementation shifts at the word level of our machine. Specifically, we shift materials by multiples of 128 bits. This coarse granularity ensures that the implementation need not even perform bit shifts; we instead simply load from/store to the correct location in memory to implement staggering.

Further low level improvements to our implementation are possible. For example, our current implementation does not *stream* the GC stacks from G to E as they are produced. Instead, we send all stacks in a batch across the network.

Compared Garbling Schemes. To compare the performance of our technique, we implemented two other GC-based techniques for handling k-out-of-n circuits:

1. *Standard Stacked Garbling.* Our primary point of comparison is stacked garbling *without* our staggering optimization. Namely, rather than using Gaussian elimination to extract k materials from k stacks, G separately garbles each of the n circuits k times.

2. *Standard GC.* For further reference, we also instantiate k-out-of-n branching using a basic Boolean circuit with no stacking optimization. This technique consumes communication/computation independent of k[6].

Evaluation Machine. We run both G and E on a single commodity laptop. The laptop runs Ubuntu 20.04 and features an Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz and 16GB RAM. Each party runs on a single thread of execution.

Network Settings. We consider two simulated network settings:

1. *LAN:* A simulated ethernet connection with 1Gbps bandwidth and 2ms round-trip latency.
2. *WAN:* A simulated wide area network connection with 100Mbps bandwidth and 20ms round-trip latency.

Networks are simulated by the `tc` program.

Benchmark. To provide a clean point of comparison, we evaluate the three GC techniques on a program that conditionally evaluates k-out-of-n different instances of SHA-256. It is, of course, unrealistic that each branch would hold the same circuit, but our goal is to capture performance characteristics only. Despite using the same circuit for each branch, we take no shortcuts. For example, we generate the cleartext circuit once for each branch and keep each separately in memory. Similarly, we garble the circuits separately for each branch.

Note that we do not expect our performance to diminish when faced with smaller circuits: our technique is lean as Gaussian elimination is implemented only with XORs. However, traditional stacked garbling must pay cost linear in the number of the conditional's inputs and outputs, and we inherit this cost. Thus, our approach is best applied when the circuits are large in comparison to the total number of inputs/outputs of the conditional.

The SHA-256 circuit has $\approx 9 \times 10^4$ AND gates.

10 Evaluation

We report experimental results obtained when running our system against both standard Stacked Garbling and standard garbled circuits without stacking.

We used all three implementations to handle k-out-of-n circuits where each circuit is SHA-256. We set n to 16 and to 128 and then varied k from 2 to $n-1$. See Section 9 for further details on the experimental setup. Figures 5 and 6 plot the results.

[6] Technically, a full standard GC implementation would have variable performance in k due to the need to multiplex the branch outputs. For simplicity, our standard GC implementation does not multiplex outputs. This omission yields a small difference in performance that is strictly in favor of the standard technique: the circuit is smaller.

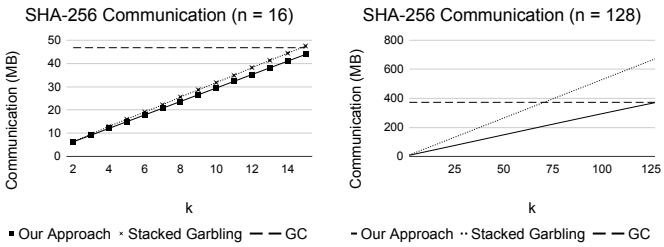

Fig. 5: S&S's and compared schemes' communication consumption as a function of k. The experiments confirm that our technique maintains the communication advantage of stacked garbling [HK20c].

Communication. Recall that our technique improves computation when evaluating k-out-of-n circuits. Figure 5 demonstrates that our method achieves this computation improvement *without* sacrificing communication.

Specifically, our technique has similar communication to the [HK20c]-based method. From the $n = 128$ chart, it is clear that our implementation of standard SGC has higher communication; we discuss why shortly. Technically, our approach's *branch* GC materials are slightly longer than those for the [HK20c]-based method because of our staggering (see e.g. Figure 1). In the last stack, each material is shifted by $k - 1$ blocks (in practice we shift each material in 128-bit blocks). Assuming all n materials are XORed into each stack, the last material is shifted by $(k - 1)(n - 1)$ blocks. This is our increase in stack length over [HK20c]. For $k = 15$, this increase is only $\approx$ 7KB and is small compared to the size of GC material even for small circuits.

We note and briefly explain the poor communication performance of the SGC protocol, which is particularly evident in the $n = 128$ case of Figure 5. This is due to the need to manage $k \cdot n$ sets of inputs and outputs, since the k SGC stacks are processed independently. I.e. our standard SGC implementation pays factor k overhead for multiplexers/demultiplexers. We note that it should be possible to reduce the SGC costs to be in line with ours; we did not implement this engineering optimization.

Wall-Clock Time. Figure 6 plots the wall-clock run-time for all three approaches and on both a LAN and a WAN. For $n = 16$, we averaged each data point over 100 runs. For $n = 128$, we averaged each data point over 10 runs.

Our experiments show that we indeed concretely improve computation in both network settings. We greatly reduce computation as compared to the standard Stacked Garbling technique, which must pay significant overhead to regarble each circuit k times. Our run-time improvement over standard garbling without stacking is less pronounced, but recall that our *communication* is significantly improved. Thus, in a sense we achieve the best of both worlds: we capture

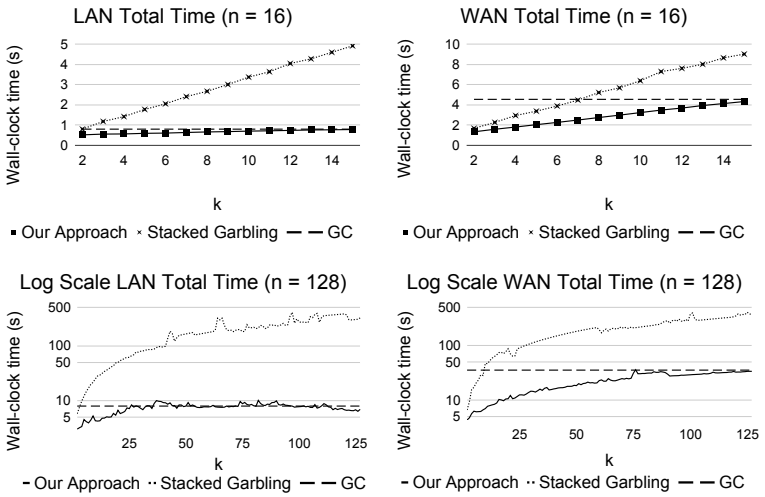

Fig. 6: S&S's and compared schemes' wall-clock runtime on both a WAN and a LAN as a function of k. As k increases, our staggering technique offers much lower computation overhead than standard Stacked Garbling. Hence, we greatly improve in terms of wall clock time. Standard GC without stacking has essentially constant performance because the parties must execute all n branches, regardless of parameter k.

the low communication utilization of standard Stacked Garbling, but without high computation.

Notice that in both settings, our performance is roughly upper bounded by the performance of standard GC without stacking. Specifically, our wall clock time approaches that of standard garbling as k approaches n. This can be explained by our choice of stacking matrix $\mathbb{A}$: as k approaches n, $\mathbb{A}$ features increasing numbers of zeros, which reduces the cost to both stack and unstack material. In the special case $n = k$, $\mathbb{A}$ features zeros everywhere except on one diagonal, where it is ones (it is a mirror of the identity matrix). Hence in this special case, our scheme and standard GC perform essentially identical actions.

We highlight specific features of the plots in Figure 6:

- **LAN wall-clock time.** On our moderately fast LAN, we improve over standard Stacked Garbling by $\approx 6.4\times$ for 15-out-of-16 circuits, and by $\approx 46.6\times$ for 127-out-of-128 circuits. We do not achieve $k\times$ improvement for two reasons. First, while we reduce the number of AES invocations from $O(n \cdot k)$ to $O(n)$, to stack and unstack materials both techniques use the same number of XOR operations. Second, both approaches consume the same amount of bandwidth, which cuts into our advantage. When we instead run the 15-out-of-16 experiment on localhost (i.e., without simulating a bandwidth limit) we achieve $\approx 10.8\times$ improvement, much closer to the $15\times$ limit. For 16-out-

of-128 circuits and on LAN, we improve over standard Stacked Garbling by $\approx 7.68\times$ and over standard GC by $\approx 4.82\times$.

- **WAN wall-clock time.** On this weaker network, bandwidth consumption becomes a greater concern. Our advantage over standard Stacked Garbling thus decreases, but our advantage over garbling without stacking increases. For 15-out-of-16 branches, we achieve $\approx 2.1\times$ speedup over standard stacked garbling; for 127-out-of-128 branches, we achieve $\approx 10.8\times$ speedup. In a 16-out-of-128 setting, we improve over standard Stacked Garbling by $\approx 7.22\times$ and over standard GC by $\approx 3.44\times$.

Acknowledgments. This work was supported in part by NSF award #1909769, by a Facebook research award, a Cisco research award, by Georgia Tech's IISP cybersecurity seed funding (CSF) award. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016.

[BMRS20] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. `https://eprint.iacr.org/2020/1410.pdf`.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. LNCS, pages 92–122. Springer, Heidelberg, 2021.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Report 2020/1446, 2020. `https://eprint.iacr.org/2020/1446`.

[GGHAK21] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Stacking sigmas: A framework to compose Σ-protocols for disjunctions. Cryptology ePrint Archive, Report 2021/422, 2021. https://eprint.iacr.org/2021/422.pdf.

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 2055–2074. ACM Press, November 2020.

[HK20b] 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.

[HK20c] 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.

[HK21] David Heath and Vladimir Kolesnikov. Logstack: Stacked garbling with o(b log b) computation. Cryptology ePrint Archive, Report 2021/531, 2021. https://eprint.iacr.org/2021/531.pdf.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 458–475. Springer, Heidelberg, August 2014.

[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.

[HKP21] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked triples - amortizing multiplication triples across conditionals. LNCS, pages 319–348. Springer, Heidelberg, 2021.

[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.

[KKW17] W. Sean Kennedy, Vladimir Kolesnikov, and Gordon T. Wilfong. Overlaying conditional circuit clauses for secure computation. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 499–528. Springer, Heidelberg, December 2017.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg, August 2014.

[KNT06] Klaus Kursawe, Gregory Neven, and Pim Tuyls. Private policy negotiation. In Giovanni Di Crescenzo and Avi Rubin, editors, *Financial Cryptography and Data Security*, pages 81–95, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[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.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, Heidelberg, August 2014.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

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

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *42nd IEEE Symposium on Security and Privacy*, 2021.

[YSWW20] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. Cryptology ePrint Archive, Report 2021/076, 2020.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.