Garbled Circuits With Sublinear Evaluator

Abida Haque[⋆], David Heath[⋆⋆], Vladimir Kolesnikov[⋆⋆⋆], Steve Lu[†], Rafail Ostrovsky[‡], and Akash Shah[§]

Abstract. A recent line of work, Stacked Garbled Circuit (SGC), showed that Garbled Circuit (GC) can be improved for functions that include conditional behavior. SGC relieves the communication bottleneck of 2PC by only sending enough garbled material for a single branch out of the b total branches. Hence, communication is sublinear in the circuit size. However, both the evaluator and the generator pay in computation and perform at least factor $\log b$ *extra* work as compared to standard GC.

We extend the sublinearity of SGC to *also* include the work performed by the GC evaluator E; thus we achieve a fully sublinear E, which is essential when optimizing for the online phase. We formalize our approach as a garbling scheme called GCWise: *GC WIth Sublinear Evaluator*.

We show one attractive and immediate application, Garbled PIR, a primitive that marries GC with Private Information Retrieval. Garbled PIR allows the GC to non-interactively and sublinearly access a privately indexed element from a publicly known database, and then use this element in continued GC evaluation.

1 Introduction

Garbled Circuit (GC) is a foundational cryptographic technique that allows two parties to jointly compute arbitrary functions of their private inputs while revealing nothing but the outputs. GC allows the parties to securely compute while using only constant rounds of communication. The technique requires that one party, the GC generator G, send to the other party, the GC evaluator E, a large "encryption" of a circuit that expresses the desired function. We refer to these circuit encryptions as GC material. The bandwidth consumed when sending GC material is typically understood to be the GC bottleneck.

Stacked Garbling [HK20a,HK21] – or Stacked GC, SGC – is a recent GC improvement that reduces bandwidth consumption for functions with conditional behavior. We review SGC in Section 3.1. In SGC, G sends material proportional

⋆ NC State, email: ahaque3@ncsu.edu
⋆⋆ Georgia Tech, email:heath.davidanthony@gatech.edu
⋆⋆⋆ Georgia Tech, email:kolesnikov@gatech.edu
† Stealth Software Technologies, Inc. email: steve@stealthsoftwareinc.com
‡ UCLA, email: rafail@cs.ucla.edu
§ UCLA, email: akashshah08@ucla.edu. Work partially done while at Microsoft Research, India.

to only the single longest branch, not to the entire circuit. Thus, SGC achieves sublinear communication for certain circuits.

Unfortunately, SGC's improved communication comes at the cost of increased computation. Let b denote the number of branches. The parties each incur at least $O(b \log b)$ computation, as compared to $O(b)$ when using standard GC [HK21].

In this work, we focus on improving the SGC computation cost of E. We mention two reasons why it is sensible to focus on E.

- **Weak E.** First, G and E may have different computational resources. We argue that E will often have weaker hardware. GC offers built-in protection against malicious E, but more sophisticated and expensive techniques are needed to protect against malicious G, see e.g., [WRK17]. Thus, the more trusted party should play G to avoid the cost of these techniques.
  We argue that in many natural scenarios, the more trusted party (e.g., a server, or a bank), is also computationally more powerful than the less trusted one (e.g., bank's client, a cell phone, an IoT device). In such scenarios, E will have weaker hardware, and E's computational power will be the bottleneck.
- **Online/offline 2PC.** Second, GC naturally allows to offload most work to an offline phase (i.e., before function inputs are available): G can construct and transmit the GC in advance. However, E can only evaluate once inputs become available in an online phase. Thus, E's computation is essentially the *only* cost in the online phase.

1.1 Our Contribution

We show that GC conditional branching can be achieved while incurring only sublinear communication *and* sublinear computation cost for E. More precisely, for a conditional with b branches, our construction requires that G send to E material of size $\tilde{O}(\sqrt{b})$ and E uses $\tilde{O}(\sqrt{b})$ computation. Our G uses $\tilde{O}(b)$ computation. Importantly, the *entire online phase* has only $\tilde{O}(\sqrt{b})$ cost.

Our construction is formalized and proved secure as a *garbling scheme* [BHR12] assuming one-way functions. (To compose our technique with Free-XOR-based schemes, we need a stronger circular correlation-robust hash function [CKKZ12].) Since it is a garbling scheme, our construction can be plugged into GC protocols. We name our garbling scheme GCWise, for GC WIth Sublinear Evaluator.

Our construction can be immediately used to build an efficient *Garbled PIR*, described next in Section 1.2. Garbled PIR allows the GC to non-interactively and sublinearly access a privately indexed element from a publicly known database, and then use this element in continued GC evaluation.

1.2 Garbled PIR

Our construction is best applied when the target conditional has high branching factor. We mention an interesting application where high branching factor naturally arises.

Suppose G and E agree on a public database with elements $(x_0, ..., x_{n-1})$. They wish to include the database as part of their GC computation by reading one of its elements. Namely, suppose the GC computes a garbled index i that is known to neither party. The parties wish to efficiently recover the value x_i *inside the GC* such that the value can be used in further computation. Such a capability is essentially Private Information Retrieval (PIR), but where the selected index and the value are compatible with GC. One can view this as the GC playing the PIR receiver and G and E jointly playing the PIR sender. We emphasize that G and E must publicly agree on the contents of the database, but they do not learn which element is accessed. For completeness, we include the following formal definition of Garbled PIR:

Definition 1 (Garbling Scheme with PIR (Garbled PIR)). *A garbling scheme [BHR12] $\mathcal{G}$ is considered a* garbling scheme with PIR *if its circuits may include the following G_{pir} gates:*

$$G_{pir}[x_0, ..., x_{n-1}](i) \mapsto x_i$$

Here G_{pir} is parameterized by the public constant array $[x_0, ..., x_{n-1}]$, and the gate input i is computed inside the evaluated circuit.

Constructing Garbled PIR from conditional branching. Efficient Garbled PIR can be immediately constructed from conditional branching. In particular, we define n conditionally composed circuits $\mathcal{C}_0, ..., \mathcal{C}_{n-1}$ such that each circuit $\mathcal{C}_i$ takes no inputs and outputs the constant x_i.

We thus obtain Garbled PIR incurring only $\tilde{O}(\sqrt{n})$ communication and $\tilde{O}(\sqrt{n})$ E computation.

Our Garbled PIR can be upgraded to store private data by using one non-black-box PRF call per access. Indeed, each x_i can be stored masked with $F_k(i)$; the GC simply accesses the i-th position, unmasks the computed PRF, and proceeds with subsequent GC evaluation.

Comparison with Garbled RAM (GRAM). It is important (and easy) to see that GRAM, introduced by [LO13], does not solve the problem of efficient Garbled PIR. Indeed, GRAM performance is *amortized* over a sequence of RAM queries. A single GRAM access will require players to jointly build and then access a superlinear data structure, a far more expensive task than a simple linear scan. Thus GRAM *does not* imply Garbled PIR with sublinear communication and E computation.

1.3 Compact 2PC and Garbled PIR

For functions with conditional behavior, we achieve communication and computation for one of the parties that is sublinear in the size of the function description (i.e., function's circuit size). We find it convenient to assign a name to this property. We call this double sublinearity *compactness*.

For example, Section 1.2 describes a compact Garbled PIR, and our garbling scheme GCWise allows to achieve compact 2PC.

1.4 High-Level Intuition for Our Approach

Let b denote the number of branches in a conditional. Rather than sending garbled material for each conditional branch, our G randomly organizes the branches into $\tilde{O}(\sqrt{b})$ *buckets* and *stacks* the branches inside each bucket. Each bucket contains $\tilde{O}(\sqrt{b})$ branches, with the constraint that each branch appears at least once (with overwhelming probability). For each bucket, G stacks the material for that bucket's branches and sends the SGC to E. This achieves sublinear $\tilde{O}(\sqrt{b})$ communication.

To achieve E's sublinear computation, we ensure that E needs to only consider a single bucket, one (possibly of several) that contains the active branch. E processes only the $\tilde{O}(\sqrt{b})$ circuits in this single bucket. The GC simply reveals to E the ID of the active bucket and the IDs of the inactive branches in it. E then unstacks the active branch material and evaluates using the remaining material.

The above description elides many details. For instance, we must route GC wire labels to 1-out-of-b circuits while maintaining sublinear communication and E computation. Additionally, we must ensure that E does not learn the identity of the active branch. We present a detailed overview of our approach in Section 4.

2 Related Work

Stacked Garbling. The most closely related works are those that developed Stacked Garbled Circuit (SGC) [Kol18,HK20b,HK20a,HK21], a GC primitive that reduces the communication cost of branching. We review the SGC technique in Section 3.1.

Our construction builds on SGC. Like prior work, we also achieve communication sublinear in the number of branches. However, we *also* achieve sublinear evaluation: our construction is compact. Prior SGC techniques are not compact.

Online-offline MPC. MPC of large functions can be expensive, and is unacceptable for certain time-sensitive (e.g., real-time) applications. One often-acceptable solution to this is to take advantage of the idle time before MPC inputs are available (the *offline* phase) by performing input-independent computation and data transfers. This often dramatically reduces the cost of the *online* phase.

MPC with preprocessing, aka online/offline MPC, is widely seen as a central setting for MPC, and is considered in many lines of work and protocol families, such as SPDZ [DPSZ12,BNO19]. Our protocol is the first one to achieve sublinear online phase for GC.

Other Garbled Circuit Optimizations. Originally, GCs required G send to E four ciphertexts per fan-in two gate.

This number of needed ciphertexts has been improved by a long line of works. While our emphasis is sublinear cost branching, not the efficiency of individual GC gates, we review such works for completeness.

- [NPS99] introduced *garbled row-reduction* (GRR3), which reduced the cost to three ciphertexts per gate.
- Much later, [KS08a] introduced the Free XOR technique which allows XOR gates to be computed without extra ciphertexts.
- [PSSW09] introduced a polynomial interpolation-based technique that uses only two ciphertexts per gate (GRR2).
- While GRR3 is compatible with Free XOR, GRR2 is not. This opened the door to further improvements: [KMR14] generalized Free XOR into "fleXOR", a technique that uses heuristics to mix GRR2 with Free XOR and GRR3.
- [ZRE15] superceded prior improvements with their half-gates technique. Half-gates consumes only two ciphertexts per AND gate and compatible with Free XOR. [ZRE15] also gave a matching lower bound in a model that seemed difficult to circumvent.
- Very recently – and quite suprisingly – [RR21] found a new approach outside the [ZRE15] lower bound model. Their technique requires only 1.5 ciphertexts per AND gate and is compatible with Free XOR.

This line of work improves the cost of individual gates; in contrast to our work, the total cost remains proportional to the circuit size.

Garbled RAM (GRAM). Most GC constructions operate in the circuit model of computation, rather than using Turing machines or RAM machines. Exceptions include the line of work on garbling schemes for RAM programs: Garbled RAM (GRAM) [LO13], outsourced RAM [GKK$^+$12], and the TM model of [GKP$^+$13]. RAM-based 2PC is motivated by the prohibitively expensive cost of generic program-to-circuit unrolling.

GRAM and our Garbled PIR are incomparable: while GRAM achieves sublinear RAM, its costs are amortized. Meanwhile, Garbled PIR is less expressive, but achieves sublinear cost without amortization.

Private Information Retrieval (PIR). Private information retrieval (PIR), introduced by Chor et al. [CGKS95,CKGS98], allows a client to retrieve an item from a public database stored at a server without revealing which item is requested. The communication complexity of PIR is sublinear in the size n of the database, and the computation of the server is linear in n. [KO97] designed a PIR scheme with communication $O(n^\epsilon)$ for an arbitrary constant ϵ; subsequent works achieved polylogarithmic communication.

We achieve Garbled PIR; i.e., private information retrieval that is compatible with GC (Section 1.2).

Compact 2PC from Fully Homomorphic Encryption (FHE). The breakthrough work on FHE by Gentry [Gen09] and Brakerski and Vaikuntanathan [BV11] can be used to achieve compact 2PC. Using FHE, one party encrypts its

Symbol	Denotation
κ	computational security parameter (e.g., 128)
$\mathcal{C}$	function/circuit
$\hat{\mathcal{C}}$	garbled circuit on $\mathcal{C}$ (uses ˆ symbol)
x, y	small Latin letters for plain inputs/outputs
X, Y	capital Latin letters for garbled inputs/outputs
G	GC generator (he/him)
E	GC evaluator (she/hers)
b	number of conditional branches
ℓ, i	number of buckets ℓ indexed by i
m, j	number of elements in a bucket indexed by j (bucket size)
α	active branch ID
β	active bucket ID
γ	the index of active instance for $\mathcal{C}_\alpha$ in active bucket B_β (see Section 5.1)
n	number of gates in a branch
S	pseudorandom seed
K	encryption key

Table 1. Table of notation.

input and sends it to the other party. The other party then computes the function *homomorphically* over these encrypted inputs and its own inputs. Hence, the communication and computation complexity of one of the parties is proportional to the size of its inputs and is independent of the size of the circuit. Despite concrete improvements, e.g., [BV11,GSW13], FHE remain expensive in practice, compared to GC.

3 Preliminaries

This section reviews stacked garbling [HK20a,HK20b] and introduces basic notation and concepts needed to understand our approach.

Notational Preliminaries. For an integer n, we use $[n]$ to denote the set $\{0, 1, \ldots, n-1\}$. PPT stands for probabilistic polynomial time. The base two logarithm of x is denoted $\log x$. We use $\stackrel{c}{=}$ to show two distributions are computationally indistinguishable. Table 1 lists various naming conventions used throughout this work.

3.1 Reducing GC Communication

A recent line of works showed that GC communication can be asymptotically improved for circuits with conditional behavior. This line began with 'Free If' [Kol18]. To reduce communication, Kolesnikov decoupled the circuit *topology*

from the garbled circuit *material*. The topology is the circuit description, describing how the gates are laid out as a graph. The material is the collection of encrypted truth tables that support secure evaluation

Free If only works when G knows the identity of the active conditional branch but ensures that E does not learn the active branch.

Building on the topology-decoupling idea, Heath and Kolesnikov showed improvements both when only E knows the active branch [HK20b] and when *neither* player knows the active branch [HK20a]. Both [HK20b] and [HK20a] consume communication proportional to only the program's longest execution path rather than to the entire circuit.

By using these stacked garbling techniques (sometimes called stacked garbled circuit, SGC), we need not send separate material for each conditional branch. Instead, a single *stacked* (via bitwise XOR) string of material can be sent for all branches. After receiving the stacked material, E is given enough information to efficiently and locally reconstruct the material for each inactive branch. This allows her to *unstack* (again, by bitwise XORing) the material for the single active branch. E can then correctly execute the active branch. By stacking the branch material, SGC greatly reduces bandwidth consumption.

[HK20b] Review. Like [HK20a], we target secure computation in the setting where neither party knows the active branch. Thus, our setting is closest to [HK20a]. While our approach is for general 2PC, our construction is more closely related to that of [HK20b], which was used to improve GC-based zero knowledge proofs [JKO13,FNO15]. The core idea given by [HK20b] does not *require* the ZK setting; it simply requires that the GC evaluator E knows the identity of each active conditional branch. Hence, we elide the ZK details and present the [HK20b] technique as one for secure 2PC.

For reference, Table 1 lists variables used to describe circuits and GCs.

Consider b branches $\mathcal{C}_0, \ldots, \mathcal{C}_{b-1}$ and let α denote the index of the active branch. Let E know α. The [HK20b] approach is as follows: G selects b PRG seeds $S_0, \ldots, S_{b-1}$ and uses each respective seed to derive all randomness used while constructing a garbling of the respective branch. Let $\hat{\mathcal{C}}_0, \ldots, \hat{\mathcal{C}}_{b-1}$ denote the b resultant GC materials (i.e., the collections of encrypted truth tables). Before [HK20b], each of these b materials would be sent to E, requiring communication proportional to the number of branches.

[HK20b] improves over this as follows: G pads the shorter materials with extra 0s until each material has the same length. G computes $\hat{\mathcal{C}} \leftarrow \bigoplus_i \hat{\mathcal{C}}_i$ and sends $\hat{\mathcal{C}}$ to E. G additionally conveys to E each seed $S_{i \neq \alpha}$ corresponding to the $b-1$ inactive branches.[1] If E were to obtain *all* GC seeds, she could use them to learn all circuit labels. This would not be secure since this would allow E to decrypt intermediate circuit values. However, it *is* secure to send seeds to E, so long as each seed is not used in an active branch [HK20b]. We ensure this is secure by using garbled gadgets to enforce that no inactive branch holds

[1] [HK20b] use oblivious transfer to convey these seeds, but they can also be encrypted according to the active branch GC labels in a GC gadget.

semantic values on its wires. Hence, there are no wire labels for E to illegally decrypt.

E uses the $b-1$ seeds to reconstruct the materials $\hat{\mathcal{C}}_{i\neq\alpha}$ and then computes $\hat{\mathcal{C}}_\alpha \leftarrow \hat{\mathcal{C}} \oplus (\bigoplus_{i\neq\alpha} \hat{\mathcal{C}}_i)$, unstacking the active branch material. E uses this active branch material and the appropriate input labels (which are conveyed separately) to evaluate the active branch.

Although we consider the setting where neither E nor G know the active branch, we leverage the above technique: we also stack GC material and reveal to E to the stacked index of the active branch. Crucially, our approach decouples the stacked index of the active branch from its index in the program. Thus, learning the former does not break security by revealing the identity of the active branch. We discuss our approach further in Section 4.

3.2 Universal and Set-Universal Circuits

To evaluate a circuit $\mathcal{C}$ inside the GC, E must both hold the material $\hat{\mathcal{C}}$ *and* know the *topology* for that circuit. However, we need to ensure that the differing topology *across* branches does not leak the identity of the active branch. This leakage can be prevented by using *universal circuits* (UC). A UC can hide the structure of the evaluated circuit.

A UC can emulate any circuit with size up to a parameterized maximum number of gates n. A UC takes as input the description of the desired circuit $\mathcal{C}$ encoded as a *programming string* c. On input x and programming string c that encodes $\mathcal{C}$, a UC $\mathcal{U}$ computes $\mathcal{U}(c,x) = \mathcal{C}(x)$.

Valiant [Val76] achieved the first UC construction, which was of size $O(n \log n)$. More recent works have improved the constant overhead of UC constructions. The current best construction [LYZ+20] achieves UCs of size $3n \log n$. A simpler construction with size $O(n \log^2 n)$ also exists and is better for small n [KS08b].

We note that UCs do not directly solve our compactness problem, since in addition to the garbled UC itself we must convey a garbling of the UC *programming string*. This programming string is proportional to the size of the UC. In general, b programming strings are needed to encode the possibility of evaluating any branch. Nevertheless, UCs are core to our approach.

Set-Universal Circuits. When we handle conditional branching, we know statically that the active branch is an element from the small set of circuits in the conditional. Thus, using a general purpose UC that emulates *any* size n circuit is overkill. *Set-universal* circuits [KKW17] construct a single circuit that can emulate any circuit from a specific *set* of circuits $\mathcal{S}$. A set-universal circuit can be less costly than a full universal circuit.

Note that for Garbled PIR (Section 1.2), the relevant set-universal circuit is incredibly simple: each "circuit" in Garbled PIR simply outputs a constant value. Hence, all such circuits already share a fixed topology and the set-universal topology is trivially constructed without overhead.

3.3 Garbled Circuit Formalization

Our approach achieves compact 2PC by using garbled circuits (GCs). Yao [Yao86] first introduced garbled circuits, with subsequent works like Lindell and Pinkas [LP09] and Bellare, Hoang, and Rogaway [BHR12] formalizing the syntax, methods, and proofs. Garbled circuit techniques are often formalized as *garbling schemes*, not as protocols. We take the same approach, formalizing our technique as a garbling scheme in the framework given by [BHR12]. In practice, the parties G and E run these algorithms as part of a protocol that uses the scheme as a black box. We present the [BHR12] garbling scheme definitions.

Definition 2 (Garbling Scheme). *A garbling scheme $\mathcal{G}$ is a tuple of algorithms:*

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

such that:

1. $(\hat{\mathcal{C}}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C})$*:* Gb *maps a function $\mathcal{C} : \{0,1\}^\ell \to \{0,1\}^m$ to a triple $(\hat{\mathcal{C}}, e, d)$ such that $\mathsf{De}(d, \cdot) \circ \mathsf{Ev}(\mathcal{C}, \hat{\mathcal{C}}, \cdot) \circ \mathsf{En}(e, \cdot) = \mathcal{C}$. We often make garbling randomness explicit via pseudorandom seed S: $(\hat{\mathcal{C}}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C}; S)$*
2. $X \leftarrow \mathsf{En}(e, x)$*:* En *maps a cleartext input $x \in \{0,1\}^\ell$ to garbled labels X by looking up labels from the* encoding string *e according to x.*
3. $y \leftarrow \mathsf{De}(d, Y)$*:* De *maps garbled output labels Y to the cleartext output y by comparing values in Y to values in the* decoding string *d.*
4. $Y \leftarrow \mathsf{Ev}(\mathcal{C}, \hat{\mathcal{C}}, X)$*:* Ev *securely evaluates a circuit $\mathcal{C}$ using its garbled material $\hat{\mathcal{C}}$ and garbled input X.*
5. $y \leftarrow \mathsf{ev}(\mathcal{C}, x)$*:* ev *evaluates the function $\mathcal{C}$ on input x in cleartext and is used to evaluate correctness. We sometimes instead write $\mathcal{C}(x)$ for simplicity.*

We formally define the security notions of a garbling scheme and show that our construction satisfies them in Section 6.

Projectivity. Our scheme only considers Boolean values and is *projective* [BHR12]. In a projective garbling scheme, each circuit wire is associated with two labels that respectively encode zero and one. Projective schemes enjoy simple definitions for En and De that map between GC labels and cleartext bits.

The encoding string e is a list of $2n$ tokens $e = (X_0^0, X_0^1, \ldots, X_{n-1}^0, X_{n-1}^1)$, two for each bit of an input $x \in \{0,1\}^n$. For a given $x = (x_0, \ldots, x_{n-1})$, $\mathsf{En}(e, x)$ selects a subvector $(X_0^{x_0}, \ldots, X_{n-1}^{x_{n-1}})$ for the encoding. Similarly, the decryption De compares output labels to the content of the decoding string d and outputs appropriate cleartext values.

3.4 Circuit Syntax

Traditionally, Boolean circuits refer to a collection of gates with specified connections. Unfortunately, this notion does not make explicit the function's conditional behavior. Therefore, we follow [HK20a] and instead refer to the above notion as

a *netlist*. Our garbling scheme (Section 5.3) handles conditionals built from a vector of netlists.

A circuit $\mathcal{C}$ is a vector of constituent netlists $\mathcal{C}_0, \ldots, \mathcal{C}_{b-1}$. As in [HK20a], we leave the syntax of netlists unspecified. This allows us to plug different low-level garbling techniques into our construction, even if the technique uses novel gates. The only restriction we place on netlists is that given a vector of netlists $\mathcal{C}_0, \ldots, \mathcal{C}_{b-1}$, it is possible to construct a universal netlist (see Section 3.2) that can be programmed (e.g., by part of its input) as any branch $\mathcal{C}_i$. By convention, the first $\lceil \log b \rceil$ bits of input to a conditional are *condition* bits that encode the active branch ID α. Semantically, on input (α, x), the conditional outputs $\mathcal{C}_\alpha(x)$.

Sequentially composed conditionals. It is often useful to sequentially compose multiple circuits, e.g., the output of one conditional is fed as input to another. While our syntax does not directly handle sequential composition, such handling can be easily laid on top of our approach, see e.g., [HK20a]. Thus, the fact that we do not further discuss sequential composition simplifies presentation but does not limit expressivity.

Nesting conditionals. We do not handle nested conditionals: it is not clear how to express a universal circuit that captures arbitrary explicit conditional branching. We note that in many cases it is possible to efficiently rewrite nested conditionals as a single top-level conditional via safe program transformations.

4 Technical Overview

In this section, we present our construction at a high level. Formal algorithms and proofs are in Sections 5 and 6. Consider b conditionally composed circuits $\mathcal{C}_{i \in [b]}$. We call these circuits *branches*. Let α denote the index of the *active branch*, i.e., the branch whose output appears at the end of the conditional. Suppose that neither G nor E knows α. Our goal is to securely compute and propagate the output of $\mathcal{C}_\alpha$ while using communication and E computation *sublinear* in the number of branches.

Standard stacked garbling. To recap Section 3.1, in standard SGC [HK20a], G constructs for each branch $\mathcal{C}_i$ a garbling $\hat{\mathcal{C}}_i$ from a seed S_i and then sends to E the stacked garbling $\bigoplus_i \hat{\mathcal{C}}_i$. At runtime, the GC conveys to E each seed $S_{i \neq \alpha}$ (via a garbled gadget programmed by G). E uses these seeds to garble each inactive branch and constructs the value $\bigoplus_{i \neq \alpha} \hat{\mathcal{C}}_i$. This value allows her to *unstack* the material for the active branch:

$$\left(\bigoplus_i \hat{\mathcal{C}}_i \right) \oplus \left(\bigoplus_{i \neq \alpha} \hat{\mathcal{C}}_i \right) = \hat{\mathcal{C}}_\alpha$$

She uses the resultant material to correctly evaluate the active branch $\mathcal{C}_\alpha$.

Unfortunately, the above procedure is not compact: E must garble *each branch*, so her work is linear in b. We adopt a different strategy.

G's handling. Instead of stacking all b garblings into a *single* stack of garbled material, G constructs *multiple* stacks. Specifically, he considers a sublinear number $\ell = \tilde{O}(\sqrt{b})$ of *buckets*, each of which is simply a collection of some of the branches. G fills each bucket with $m = \tilde{O}(\sqrt{b})$ branches via a garbled gadget called the *bucket table* (see Section 5.1). The bucket table ensures that each branch appears at least once with overwhelming probability. For each bucket B_i, G garbles the m constituent branches using m distinct seeds and stacks the resultant material. G separately sends to E the stacked material for each bucket B_i. At runtime, E will consider only *one* of these buckets. Since the considered bucket holds only $\tilde{O}(\sqrt{b})$ branches, E's work is sublinear in b.

Terminology. In our construction, a particular branch may be stacked more than once. Indeed, each branch may appear in multiple buckets and even multiple times within the *same bucket.* Each copy of a branch is called an *instance.* There are more instances than there are branches and (with overwhelming probability) there exists at least one instance of each branch. All instances in the same bucket are called *siblings.*

E need not evaluate all instances: many are *dummies* that prevent E from learning the active branch ID. At runtime, E will evaluate a garbling of exactly one instance of branch $\mathcal{C}_\alpha$. We call this evaluated instance the *active* instance. The active instance resides in a bucket that we call the *active bucket*; we denote the active bucket ID by β.

E's handling of buckets. Recall that the GC computes the value α and that E holds stacked material for each bucket. The garbled material for the active instance is in the active bucket B_β. We proceed as follows: The GC reveals to E the following information via the *bucket table*:

1. The identity of the active bucket β.
2. The identity of the active instance's $m - 1$ siblings: i.e., which inactive branches are in B_β.
3. The $m - 1$ seeds used to garble the active instance's siblings.

E, crucially, is *not* given information about any inactive buckets $B_{i \neq \beta}$ and is *not* told the identity of the active instance. We show that the above information can be compactly and securely computed by our carefully arranged bucket table gadget (see Section 5.1).

With this information, E garbles each sibling instance and unstacks the active instance's material. Crucially, our bucket table ensures that the branches within a single bucket are sampled *with replacement*, so even learning that branch $\mathcal{C}_j$ is a sibling of the active instance does not allow E to rule out the fact that $\mathcal{C}_j$ might be the active branch. From here, we would like E to evaluate the active instance. However, one important problem remains: to evaluate, E needs both the active instance's material (which she has) *and* the active branch *topology.* As discussed so far, E cannot learn this topology, since this would immediately imply the identity of the active branch.

Universal topology. To avoid the above problem, we ensure each branch $\mathcal{C}_i$ uses *the same topology*. We achieve this by expressing each branch as the programming of a *universal circuit* (UC) (see Section 3.2). Since each branch has the same topology, E can evaluate the active branch without learning its identity.

This raises a question: Why not instead simply use one UC to directly express the conditional instead of stacking garbled material? The crucial problem with using UCs for conditional branching is that E must somehow obtain a garbled *programming string* corresponding to the active branch. Standard techniques for conveying 1-out-of-b programming strings require communication proportional to b, and so are not compact.

In our approach, programming strings are sent efficiently: G incorporates the programming string directly into each garbling. Thus, when E unstacks, she obtains a garbling with the proper programming for the active branch, but without learning the active branch ID and without needing to consider all b possible functions.

Summary of our approach.

- G and E agree on a circuit $\mathcal{U}$ that is universal to each branch $\mathcal{C}_i$.
- G considers $\tilde{O}(\sqrt{b})$ buckets and fills each bucket with $\tilde{O}(\sqrt{b})$ branch IDs.
- For each instance, G accordingly programs $\mathcal{U}$ and garbles programmed $\mathcal{U}$. For each bucket, G stacks the $\tilde{O}(\sqrt{b})$ materials.
- G sends $\tilde{O}(\sqrt{b})$ materials to E. The materials include the stacked garbling for each bucket and the garbled gadgets, including the *bucket table*. The bucket table tells E how to unstack the active instance.
- E evaluates the bucket table and learns the active bucket, the identities of the siblings of the active instance, and seeds for these siblings.
- E considers *only* the active bucket, garbles the siblings, unstacks the active instance material, and evaluates the active instance.

By running the above high-level procedure, E evaluates a conditional with b branches, but while using only $\tilde{O}(\sqrt{b})$ communication and computation. The technique does require a garbled bucket table gadget (and a demultiplexer and multiplexer gadget), but we show that the gadgets can be constructed with size sublinear in the number of branches. Hence, G and E obliviously execute a conditional while using only sublinear communication and E computation: we achieve compact 2PC.

5 Our Construction

In this section, we present our technique in detail. We start by describing the bucket table gadget. Then we introduce our multiplexer (mux) and demultiplexer (demux) gadgets. One key idea (similar to SGC) is different parts of the circuit are garbled with different seeds. This creates the problem that different circuit wires are associated with two different GC *labels*. Garbling even the same circuit starting from a different seed will result in different GC wire labels: we say

that different GCs have different *vocabularies*. The mux/demux gadgets handle a problem of *vocabulary translation* needed to evaluate one out of many different garbled circuits.

Finally, we combine our gadgets and the high level ideas from Section 4 into a *garbling scheme* [BHR12]. Section 6 then proves this garbling scheme is secure.

5.1 Bucket Table Gadget

In this section, we formalize the *bucket table* gadget, which is the garbled gadget that tells E the information needed to evaluate the active branch $\mathcal{C}_\alpha$. Given a garbled encoding of the branch id α, the bucket table gives the following information to E:

- The active bucket's identity, β.
- The identity of the siblings of $\mathcal{C}_\alpha$ in B_β.
- $m-1$ seeds corresponding to the garbling of each sibling.
- A *bucket key* K_β corresponding to the active bucket. Each bucket's garbling is encrypted by a distinct key that ensures E can only view the the active bucket's garbling.

To implement the bucket table using only sublinear work, we use a key insight: we only need to sample enough randomness for *one bucket* as we can reuse this sampled randomness across buckets.

In our bucket table gadget, we sample m uniform offsets $\delta_i \in [b]$. These m offsets comprise the choices of branches for each bucket. Specifically, we place each branch id $(\delta_i + j) \bmod b$ at the i^{th} index of bucket B_j. That is, we use the same m random offsets for each bucket but apply a deterministic per-bucket linear shift. Table 2 depicts the assignment of branches to buckets. As the random choices are made *with replacement*, a branch may appear more than once in a bucket. This approach is similar to a technique used to achieve PIR with sublinear online time [CK20].

Besides assigning branches to buckets, the bucket table also samples m garbling seeds S_i and ℓ encryption keys K_j uniformly at random. Each seed S_i will be used to garble the i^{th} branch in every bucket B_j. Key K_j will be used to encrypt the stacked material corresponding to bucket B_j (see Section 5.3).

At runtime, the bucket table takes as argument a garbling of the active branch id α and computes, based on the list δ_i, the identity of the active bucket β *and* an index γ within the active bucket that holds the active instance. In this procedure, we must ensure that E learns no information about α. Since our bucket table will often include multiple instances corresponding to active branch $\mathcal{C}_\alpha$, we must choose among these instances uniformly. Moreover, we must make this choice using work sublinear in the number of branches. We define the bucket table procedure below:

1. Identify each instance of the active branch $\mathcal{C}_\alpha$. To perform this in sublinear time, iterate over the list of offsets $\delta_{i\in[m]}$ and build a list $\mathsf{instances}$ of those

indices i for which some bucket holds a garbling of $\mathcal{C}_\alpha$ at position i. The $\mathsf{instances}$ list can be built by computing

$$\gamma_i = (\alpha - \delta_i) \bmod b, \text{ for } i \in [m].$$

If $\gamma_i \in [\ell]$, then set $\mathsf{instances}[i] = 1$, indicating that there is a bucket that holds an instance of $\mathcal{C}_\alpha$ at a position corresponding to γ_i; else set $\mathsf{instances}[i] = 0$ to indicate that there does not exist a bucket id j such that $(\delta_i + j) \bmod b = \alpha$.

2. Select a single active instance by uniformly sampling among the non-zero indices of $\mathsf{instances}$. This can be achieved as follows (1) Compute the hamming weight $\mathsf{HW}(\mathsf{instances})$, (2) Select a large uniform value r (this can be done outside the GC by G), (3) Compute $t = r \bmod \mathsf{HW}(\mathsf{instances})$ which, for $r \gg \mathsf{HW}(\mathsf{instances})$ is statistically indistinguishable from uniform, and (4) Linearly scan the list δ_i to select the t^{th} non-zero index of $\mathsf{instances}$ which is denoted by γ. Select the value δ_γ via a linear scan over each δ_i.
3. Identify the active bucket $\beta \leftarrow (\alpha - \delta_\gamma) \bmod B$. The index of the active instance within the active bucket is γ.
4. Compute each sibling $y_{i \neq \gamma} = \delta_{i \neq \gamma} + \beta$, each sibling seed $S_{i \neq \gamma}$, and the bucket key K_β: each of these values is computed by linearly scanning lists of offsets $\delta_{i \in [m]}$, garbling seeds $S_{i \in [m]}$, and encryption keys $K_{j \in [\ell]}$ respectively, with respect to β and γ.

Let $\mathcal{C}_{\text{bt}}$ denote the circuit that computes the above procedure. To summarize, $\mathcal{C}_{\text{bt}}$ takes as input the active branch id α and outputs the active bucket id $\beta \in [\ell]$, the index of active instance in that bucket $\gamma \in [m]$, the siblings of the active instance $y_{i \neq \gamma}$, the seeds $S_{i \neq \gamma}$, and the encryption key K_β. Observe that $\mathcal{C}_{\text{bt}}$ has size $\tilde{O}(\ell + m)$ as it only consists of linear scans of lists of length ℓ and m.

Let $\mathsf{BT.Gb}$ denote the procedure that takes as input lists of offsets $\delta_{i \in [m]}$, garbling seeds $S_{i \in [m]}$, encryption keys $K_{j \in [\ell]}$, the GC vocabulary for the possible active branch labels $\hat{\gamma}$ and constructs a garbled circuit $\hat{\mathcal{C}}_{\text{bt}}$ for circuit $\mathcal{C}_{\text{bt}}$. Let $\mathsf{BT.Ev}$ denote the evaluation procedure that takes as input the garbled circuit $\hat{\mathcal{C}}_{\text{bt}}$ and an encoding of the active branch id $\hat{\alpha}$ and outputs $\mathcal{C}_{\text{bt}}(\alpha)$.

We define an additional subprocedure $\mathsf{Proc_{Bkt}}$ which G uses to sample necessary random values used in the bucket table. Specifically, $\mathsf{Proc_{Bkt}}$ samples (1) samples the m offsets $\delta_{i \in [m]}$, (2) assigns branches to each of the buckets, (3) samples the m garbling seeds $S_{i \in [m]}$, and (4) samples ℓ encryption keys $K_{j \in [\ell]}$. $\mathsf{Proc_{Bkt}}$ is described in Figure 1. In Lemma 1, we prove that by setting ℓ and m to $\tilde{O}(\sqrt{b})$, all branches appear with overwhelming probability. Hence, the size of circuit $\mathcal{C}_{\text{bt}}$ is $\tilde{O}(\sqrt{b})$.

Lemma 1. *If $\ell = \tilde{O}(\sqrt{b})$ and $m = \tilde{O}(\sqrt{b})$, then the bucket table (Figure 1) places each branch $\mathcal{C}_\eta$ (for $\eta \in [b]$) into a bucket with overwhelming probability.*

Proof. Let $m = \sqrt{b}\kappa$ and $\ell = \sqrt{b}$. We analyze the probability that branch $\eta \in [b]$ does not belong to any of the ℓ buckets $B_{j \in [\ell]}$. Let $\gamma_j = \eta - \delta_j \bmod b$, where

$\mathsf{Proc}_{\mathsf{Bkt}}(m, \ell, b)$:
1: Uniformly sample m offsets $\delta_{i\in[m]} \in [b]$.
2: Uniformly sample m garbling seeds $S_{i\in[m]}$ and ℓ encryption keys $K_{j\in[\ell]}$.
3: For each $j \in [\ell]$, build bucket $B_j = [\delta_i + j]_{i\in[m]}$.
4: **Return** $\delta_{i\in[m]}, S_{i\in[m]}, B_{j\in[\ell]}, K_{j\in[\ell]}$.

Fig. 1. Procedure to construct bucket table, $\mathsf{Proc}_{\mathsf{Bkt}}$.

Bucket B_0	δ_0	$\ldots$	δ_i	$\ldots$	δ_{m-1}
$\vdots$					
Bucket B_j	$\delta_0 + j$	$\ldots$	$\delta_i + j$	$\ldots$	$\delta_{m-1} + j$
$\vdots$					
Bucket $B_{\ell-1}$	$\delta_0 + \ell - 1$	$\ldots$	$\delta_i + \ell - 1$	$\ldots$	$\delta_{m-1} + \ell - 1$

All arithmetic operations are in $\mathbb{Z}_b$.

Table 2. The Bucket Table assigns branches to buckets. Each branch id $(\delta_i + j) \bmod b$ is placed at index i of bucket B_j.

$j \in [m]$. Since each δ_j is uniform at random,

$$\Pr[\gamma_j \notin [\ell]] = 1 - \frac{\ell}{b}.$$

Moreover, since each δ_j is independent:

$$\Pr[\eta \notin B_1 \wedge \cdots \wedge \eta \notin B_\ell] = \Pr[\gamma_1 \notin [\ell] \wedge \cdots \wedge \gamma_m \notin [\ell]]$$
$$= \left(1 - \frac{\ell}{b}\right)^m = \left(1 - \frac{\sqrt{b}}{b}\right)^{\sqrt{b}\kappa} = \frac{1}{e^\kappa} = \mathsf{negl}(\kappa).$$

□

5.2 Demultiplexer and Multiplexer

The bucket table allows E to unstack material for the active instance γ in the active bucket β, but it does not suffice to route inputs to (resp. outputs from) the active instance. E needs more information to evaluate the active branch $\mathcal{C}_\alpha$.

In general, the conditional composition of b branches can occur in the middle of a circuit, with sequentially composed circuits occurring before and after the conditional. To route input and output GC labels to enter and exit the conditional, we design an additional demultiplexer (demux) and multiplexer (mux) gadget.

The demux and mux map the vocabulary of the surrounding circuit (i.e., the circuit that holds the conditional branch) to the vocabulary of each instance. Both the demux and mux operate at the level of a particular bucket: they translate the vocabulary of the surrounding circuit to the vocabulary of one instance in that bucket. Thus, the demux and mux are compact, since their size is proportional to the number of elements in a bucket. We can reuse the same demux and mux across all buckets, and hence our vocabulary translation for the full conditional is compact.

The demultiplexer computes the following function for each input wire to the conditional x and each bucket index i:

$$\mathsf{demux}(x, i, \gamma) = \begin{cases} x & \text{if } i = \gamma \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ indicates that the demultiplexer makes no promise if the instance is inactive. In other words, the demultiplexer delivers valid labels to the active instance, but not to any inactive instance. In the GC, the demux is an encrypted truth table that maps each input label X to a corresponding label X_i for each i^{th} instance. The truth table is encrypted by the GC labels that encode γ such that E can *only* decrypt valid input labels for the active instance X_γ, and not for any inactive instance.

Similarly, the multiplexer computes the following simple function that selects outputs from the active instance

$$\mathsf{mux}(y_1, ..., y_b, \gamma) = y_\gamma$$

In the GC, the mux is, again, built by encrypted truth tables that map each output label from each i^{th} instance Y_i to an output label for the surrounding circuit Y. Again, this truth table is encrypted according to GC labels that encode γ such that E can *only* translate outputs labels Y_γ of the active instance, not any inactive instance.

Both the demultiplexer and multiplexer can be built as simple garbled gadgets that use encrypted truth tables, like techniques used in [HK20a]. However, one crucial observation ensures both gadgets are compact: it is sufficient to sample only m total garbling seeds S_i. These same m seeds can be reused across the ℓ buckets. Because the buckets reuse the seeds and every circuit uses the same universal topology, there are only m total vocabularies: each i^{th} garbling in a given bucket is garbled starting from the ith seed S_i, so the i^{th} circuits across all buckets share the same vocabulary. This fact means that the demultiplexer (resp. multiplexer) need only translate to (resp. from) m different vocabularies, and so is compact.

Our construction uses four procedures:

- $\mathsf{demux.Gb}$ garbles the demux. It takes as arguments (1) the input vocabulary for each i^{th} instance e_i and (2) the GC label vocabulary for the active instance id γ. It outputs (1) the input vocabulary from the overall conditional e and (2) a garbled circuit $\hat{\mathcal{C}}_{\text{dem}}$ that encodes the demux procedure.

demux.Gb samples the input encoding string e uniformly, with the exception that each pair of labels for a given label have differing least significant bits.
- demux.Ev evaluates the demux. It takes as arguments (1) a GC $\hat{\mathcal{C}}_{\text{dem}}$, (2) GC labels that encode the active branch id γ, and (3) surrounding circuit inputs X. It outputs inputs for the active instance X_γ.
- mux.Gb garbles the mux. It takes as arguments (1) the output vocabulary for each i^{th} instance d_i and (2) the GC label vocabulary for the active instance id γ. It outputs (1) the output vocabulary from the overall conditional d and (2) a garbled circuit $\hat{\mathcal{C}}_{\text{mux}}$ that encodes the mux procedure. mux.Gb samples the output decoding string d uniformly, with the exception that each pair of labels for a given label have differing least significant bits.
- mux.Ev evaluates the mux. It takes as arguments (1) a GC $\hat{\mathcal{C}}_{\text{mux}}$, (2) GC labels that encode the active branch id γ, and (3) GC output labels from the active instance Y_γ. It outputs output labels for the overall conditional Y.

5.3 Our Garbling Scheme

Following our syntax from Definition 2, we construct our garbling scheme GCWise:

Construction 1 (GCWise Garbling Scheme)**.** *Let* Base *be an underlying garbling scheme that satisfies the GC properties of* correctness*,* obliviousness*,* privacy*,* authenticity*, and* sequential composability *(see Section 3.3). Then* GCWise *is the five tuple of algorithms:*

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

as defined in Figure 2.

Construction 1 supports compact 2PC for conditional circuits. Specifically, for a conditional with b branches each with n gates, GCWise.Gb outputs a material of size $\tilde{O}(\sqrt{b} \cdot n)$ and GCWise.Ev runs in $\tilde{O}(\sqrt{b} \cdot n)$ time.

Construction 1 is the relatively straightforward formalization of our technique as explained in Section 4. The key algorithmic details arise from our garbled gadgets, particularly the bucket table, and were formalized in Sections 5.1 and 5.2.

We note some of the interesting details of Construction 1:

- Our garbling scheme is *projective* [BHR12]. As discussed in Section 3.3, a projective garbling scheme has a simplified input and output vocabulary, so we can use standard algorithms to implement GCWise.En and GCWise.De. We simply reuse the encoding and decoding algorithms of Base.
- Our algorithms GCWise.Gb and GCWise.Ev formalize the core of our approach as explained in Section 4.
- Notice that we call Base.Gb with an additional seed argument. Recall from Definition 2 that this denotes that we configure the randomness of the procedure with an explicit seed.

– Our scheme passes the universal circuit $\mathcal{U}$ to both $\mathsf{Base.Gb}$ and $\mathsf{Base.Ev}$. In the former case we write $\mathcal{U}[\mathcal{C}_i]$ to denote that Gb hardcodes the programming string inputs based on $\mathcal{C}_i$. This ensures that the garbled material $\hat{\mathcal{C}}_i$ includes the garbled programming string for the UC. In the latter case, therefore, E can evaluate $\mathcal{U}$ without knowing $\mathcal{C}_\alpha$.

6 Security

In this section, we first introduce the security notions of a garbling scheme [BHR12], then formally prove that Construction 1 satisfies these notions.

Informally, the GC security notions are as follows:

– Privacy: $(\hat{\mathcal{C}}, X, d)$ reveals no more about x than $\mathcal{C}(x)$. Formally, there must exist a simulator Sim_{pr} that takes the input $(1^\kappa, \mathcal{C}, \mathcal{C}(x))$ and produces an output that is indistinguishable from $(\hat{\mathcal{C}}, X, d)$.
– Obliviousness: $(\hat{\mathcal{C}}, X)$ reveals no information about x. Formally, there must exist a simulator Sim_{ob} that takes input $(1^\kappa, \mathcal{C})$ and produces an output that is indistinguishable from $(\hat{\mathcal{C}}, X)$.
– Authenticity: Given only $(\hat{\mathcal{C}}, X)$ no adversary should be able to produce $Y' \neq \mathsf{Ev}(\hat{\mathcal{C}}, X)$ such that $\mathsf{De}(d, Y') \neq \perp$ except with negligible probability.

The games for privacy and obliviousness are illustrated in Fig. 3.

Definition 3 (Correctness). *For $\mathcal{C} \in \{0,1\}^*$, $\kappa \in \mathbb{N}$, and $x \in \{0,1\}^n$, and $(\hat{\mathcal{C}}, e, d) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C})$:*

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

Definition 4 (Obliviousness). *A garbling scheme $\mathcal{G}$ is* oblivious *if for all λ large enough, there exists a polynomial-time simulator Sim such that for any PPT adversary $\mathcal{A}$:*

$$Pr[\mathsf{ObvSim}^{\mathcal{A}}_{\mathcal{G},\mathsf{Sim}}(1^\kappa) = 1] \leq \mathsf{negl}(\kappa).$$

Definition 5 (Privacy). *A garbling scheme $\mathcal{G}$ is* private *if for all λ large enough, there exists a polynomial-time simulator Sim such that for any PPT adversary $\mathcal{A}$:*

$$Pr[\mathsf{PrivSim}^{\mathcal{A}}_{\mathcal{G},\mathsf{Sim}}(1^\kappa) = 1] \leq \mathsf{negl}(\kappa).$$

Definition 6 (Authenticity). *A garbling scheme $\mathcal{G}$ is* authentic *if for all sufficiently large λ and for any polynomial time adversary $\mathcal{A}$:*

$$\Pr[\mathcal{A} \textit{ wins } \mathsf{AuthGame}(1^\lambda)] \leq \mathsf{negl}(\lambda)$$

$\mathsf{GCWise.Gb}(1^\kappa, \mathcal{C}_0, ..., \mathcal{C}_{b-1})$

$\delta_{i\in[m]}, S_{i\in[m]}, B_{j\in[\ell]}, K_{j\in[\ell]} \leftarrow \mathsf{Proc}_{\mathsf{Bkt}}(m, \ell, b)$
$(\hat{\alpha}, \hat{\gamma}, \hat{\mathcal{C}}_{\mathrm{bt}}) \leftarrow \mathsf{BT.Gb}(\delta_{i\in[m]}, S_{i\in[m]}, K_{j\in[\ell]})$
for $j \in [\ell]$ **do**
  $\mathcal{M}_j \leftarrow 0$
  for $i \in [m]$ **do**
    $\mathrm{ix} \leftarrow B_j[i]$
    $(\hat{\mathcal{C}}_{\mathrm{ix}}, e_i, d_i) \leftarrow \mathsf{Base.Gb}(1^\kappa, \mathcal{U}[\mathcal{C}_{\mathrm{ix}}]; S_i)$
    $\mathcal{M}_j \leftarrow \mathcal{M}_j \bigoplus \hat{\mathcal{C}}_{\mathrm{ix}}$
  $\tilde{\mathcal{M}}_j \leftarrow \mathsf{Encrypt}(\mathcal{M}_j, K_i)$
$(e', \hat{\mathcal{C}}_{\mathrm{dem}}) \leftarrow \mathsf{demux.Gb}(\hat{\gamma}, e_{i\in[m]})$
$(d, \hat{\mathcal{C}}_{\mathrm{mux}}) \leftarrow \mathsf{mux.Gb}(\hat{\gamma}, d_{i\in[m]})$
$e \leftarrow (\hat{\alpha}, e')$
$\hat{\mathcal{C}} \leftarrow (\tilde{\mathcal{M}}_{j\in[\ell]}, \hat{\mathcal{C}}_{\mathrm{bt}}, \hat{\mathcal{C}}_{\mathrm{dem}}, \hat{\mathcal{C}}_{\mathrm{mux}})$
return $(\hat{\mathcal{C}}, e, d)$

$\mathsf{GCWise.Ev}(\mathcal{C}_0, ..., \mathcal{C}_{b-1}, \hat{\mathcal{C}}, X)$

Parse $\hat{\mathcal{C}}$ as $(\tilde{\mathcal{M}}_{j\in[\ell]}, \hat{\mathcal{C}}_{\mathrm{bt}}, \hat{\mathcal{C}}_{\mathrm{dem}}, \hat{\mathcal{C}}_{\mathrm{mux}})$
Parse X as $(\hat{\alpha}, X')$
$(\beta, \gamma, \hat{\gamma}, (B_\beta \setminus \gamma), S_{i\in[m]\setminus\gamma}, K_\beta) \leftarrow \mathsf{BT.Ev}(\hat{\mathcal{C}}_{\mathrm{bt}}, \hat{\alpha})$
$X_\gamma \leftarrow \mathsf{demux.Ev}(\hat{\mathcal{C}}_{\mathrm{dem}}, \hat{\gamma}, X')$
$\mathcal{M}_\beta \leftarrow \mathsf{Decrypt}(\tilde{\mathcal{M}}_\beta, K_\beta)$
for $i \in [m] \setminus \gamma$ **do**
  $\mathrm{ix} \leftarrow B_\beta[i]$
  $(\hat{\mathcal{C}}_{\mathrm{ix}}, \cdot, \cdot) \leftarrow \mathsf{Base.Gb}(1^\kappa, \mathcal{U}[\mathcal{C}_{\mathrm{ix}}]; S_i)$
  $\mathcal{M}_\beta \leftarrow \mathcal{M}_\beta \oplus \hat{\mathcal{C}}_{\mathrm{ix}}$
$\hat{\mathcal{C}}_\alpha \leftarrow \mathcal{M}_\beta$
$Y_\gamma \leftarrow \mathsf{Base.Ev}(\mathcal{U}, \hat{\mathcal{C}}_\alpha, X_\gamma)$
$Y \leftarrow \mathsf{mux.Ev}(\hat{\mathcal{C}}_{\mathrm{mux}}, \hat{\gamma}, Y_\gamma)$
return Y

Fig. 2. Our garbling scheme GCWise. Recall from Section 3.4 that our scheme considers the conditional composition of b netlists. Let $\mathcal{U}$ be a circuit universal to $\mathcal{C}_0, ..., \mathcal{C}_{b-1}$; $\mathcal{U}[\mathcal{C}_i]$ denotes hardcoding the programming string of $\mathcal{U}$ according to the circuit description $\mathcal{C}_i$. Since GCWise is a projective garbling scheme [BHR12], procedures GCWise.En and GCWise.De are standard constructions that implement straightforward mappings between cleartext Boolean values and GC labels (see Section 3.3). The semantic function GCWise.ev gives the straightforward semantics of a conditional and is defined as follows: $\mathsf{GCWise.ev}(\mathcal{C}_0, ..., \mathcal{C}_{b-1}, \alpha, x) \mapsto \mathcal{C}_\alpha(x)$. Our construction uses our three garbled gadgets: the bucket table BT (see Section 5.1) as well as the demux and mux (see Section 5.2). Our scheme is parameterized over an underlying garbling scheme Base which we use to handle the individual conditional branches.

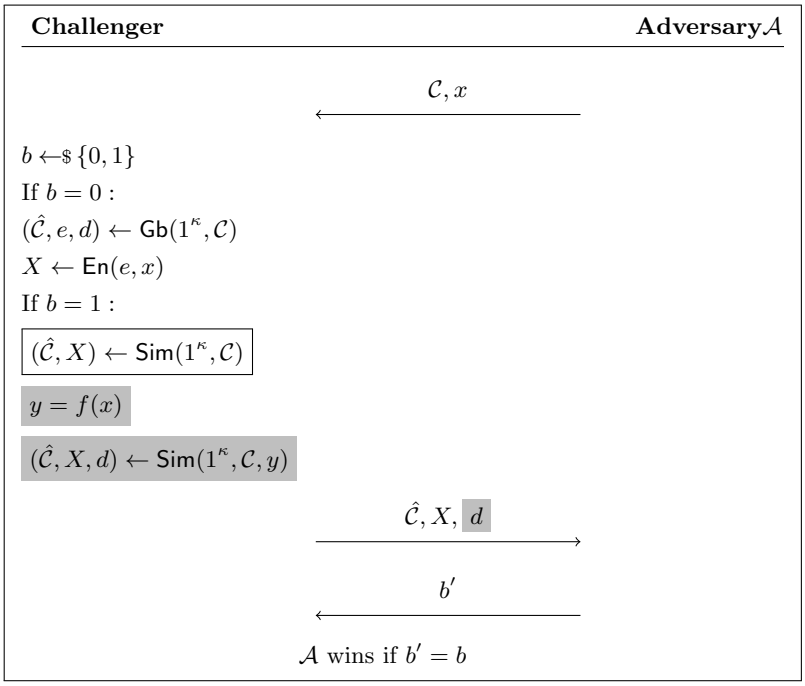

Fig. 3. Games for $\mathsf{ObvSim}^{\mathcal{A}}_{\mathcal{G},\mathsf{Sim}}$ and $\mathsf{PrivSim}^{\mathcal{A}}_{\mathcal{G},\mathsf{Sim}}$. The steps in boxes only apply to ObvSim, and the highlighted steps only apply to $\mathsf{PrivSim}$. Unmarked text means the steps appear in both games.

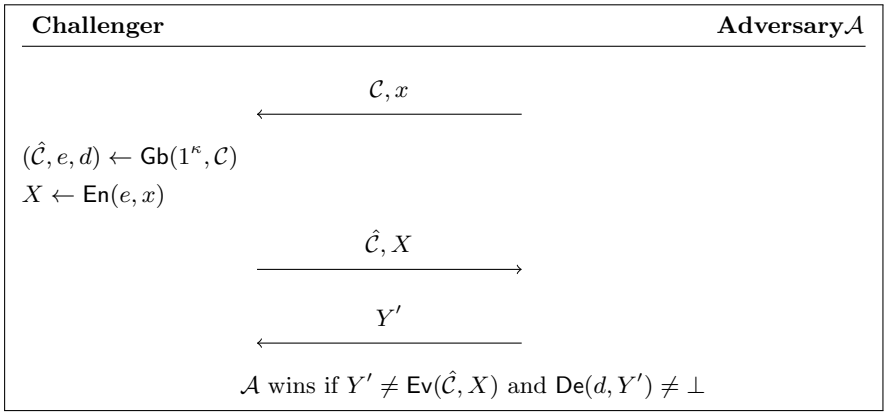

Fig. 4. Game for $\mathsf{AuthGame}^{\mathcal{A}}_{\mathcal{G}}$.

Sequential Composability. As explained in Section 3.4, we do not directly manage the low level handling of individual gates. We instead adopt an approach given by [HK20a], where we leave the handling of netlists to a parameterized *underlying* garbling scheme. Arbitrary garbling schemes are not candidates for the underlying scheme because they do not export the format of their GC labels. To interface with the underlying scheme, we need to build *garbled gadgets* such that we can route wire labels into and out of conditional branches. Therefore, we define a concept of *sequentially composable* garbling schemes, a weakening of the *strong stackability* property given by [HK21]. Informally, sequential composability requires the garbling scheme to export the format of its labels such that they can be directly manipulated (i.e., used as PRF keys) by higher level garbling schemes. A sequentially composable scheme is projective and has a color and key function $\mathsf{colorPart}$ and $\mathsf{keyPart}$. Many traditional garbling schemes, such as the classic 4-row Yao scheme, or the more recent half-gates [ZRE15], are sequentially composable or can be trivially adjusted (in a formal sense, meaning that only syntactic changes are needed) to meet the requirements.

As with [HK20a], we use the output labels of the underlying scheme as keys in subsequent garbled gadgets. We explain these gadgets in Section 5.2, but basically, they are implemented as garbled rows. The $\mathsf{keyPart}$ procedure gives us a key for each label. The $\mathsf{colorPart}$ procedure tells us the bits to instruct E as to which garbled row to decrypt. We 'split' each output label into a key and a color.

Definition 7 (Sequential Composability). *A garbling scheme is* sequentially composable *if:*

1. *The scheme is projective, including with respect to decoding. I.e., the output decoding string d is a vector of pairs of labels, and the procedure $\mathsf{De}(d, Y)$ is a simple comparison that, for each output label $Y_i \in Y$, computes the following output bit:*

$$\begin{cases} 0 & \text{if } Y_i = d_i^0 \\ 1 & \text{if } Y_i = d_i^1 \\ \bot & \text{otherwise} \end{cases}$$

2. *There exists an efficient deterministic procedure* $\mathsf{colorPart}$ *that maps bitstrings to $\{0,1\}$ such that for all projective label pairs $X^0, X^1 \in d$:*

$$\mathsf{colorPart}(X^0) \neq \mathsf{colorPart}(X^1)$$

   for the projective label pairs of the garbling scheme.
3. *There exists an efficient deterministic function* $\mathsf{keyPart}$ *that maps bitstrings to $\{0,1\}^\kappa$. Let k be the concatenation of the result of applying* $\mathsf{keyPart}$ *to each label in the output decoding string d. Let $R \in_\$ \{0,1\}^{|k|}$ be a uniform string:*

$$k \stackrel{c}{=} R$$

Note that the definition discusses the output decoding string d. Normally, d is used at the final layer of the GC to reveal outputs to E. *This is not our intent here.* We will not reveal the underlying scheme's d to E. Rather, we use d as a hook by which our garbling scheme can syntactically manipulate the labels of the underlying scheme to glue the output of the underlying scheme with the next layer of gates.

Free XOR [KS08a] based schemes (e.g., [ZRE15]) might appear to violate sequential composability: in Free XOR, each pair of internal wire labels is related by single global constant. Note, Free XOR-based schemes must *not* use the global constant as an offset for the output decoding string d, since otherwise the scheme would clearly fail to satisfy privacy (Definition 5). To resolve this issue, Free XOR-based schemes usually apply a hash function H to break correlation between labels inside the De function. To meet the letter of Definition 7, we simply push these hash function calls into the Ev function. Thus, these schemes effectively do generate output labels that are indistinguishable from uniformly random strings (i.e., that meet requirement 3 of Definition 7). This syntactic reinterpretation does not imply semantic change in [KS08a,ZRE15].

6.1 Proofs

In this section, we prove that GCWise satisfies the above garbled circuit security notions. Recall that Base is the underlying garbling scheme used to handle the content of individual branches. Our theorems have the form "If Base satisfies property X and sequential composability (Definition 7), then GCWise satisfies property X." The additional assumption of sequential composability is needed to so our garbling scheme can manipulate the GC labels of Base. Specifically, the sequential composability property allows us to use the colorPart and keyPart procedures to construct encrypted truth tables.

We first prove a lemma that our scheme is itself sequentially composable. This lemma can be used to embed GCWise inside a higher level scheme such that, for example, many conditionals can be sequentially composed (see discussion in Section 3.4).

Lemma 2. GCWise *is sequentially composable (Definition 7).*

Proof. The sequential composability of our scheme follows trivially from the definition of mux.Gb (Section 5.2). This procedure samples a uniform projective decoding string d with the constraint that the least significant bit of each label pair differs. Thus, we can use the least significant bit of each label as its color and the remaining bits as the key. □

We next prove our scheme satisfies the properties of *correctness*, *authenticity*, *obliviousness*, and *privacy*. By satisfying these properties we ensure that our scheme can be securely plugged into GC protocols that use garbling schemes as a black box.

Theorem 1. *If the underlying garbling scheme* Base *is correct and sequentially composable then* GCWise *is correct.*

Proof. Correctness follows from (1) the discussion in Section 4, (2) the correctness of Base, and (3) the correctness of our garbled gadgets, as implied by the sequential composability of Base.

Let $\mathcal{C}_0, ..., \mathcal{C}_{b-1}$ be a vector of arbitrary circuits. Each branch $\mathcal{C}_i$ is garbled using Base. By construction, the $\mathcal{C}_i$ is stacked in buckets, and E obtains the material only for the active branch $\mathcal{C}_\alpha$.

Going in steps, the bucket table (Section 5.1) first reveals to E the information needed to extract material for the active instance:

- The identity of the active bucket, β.
- The identity of the siblings of $\mathcal{C}_\alpha$ in B_β.
- $m-1$ seeds corresponding to the garbling of each sibling.
- A *bucket key* K_β corresponding to the active bucket.

E uses this information to decrypt and unstack the material $\hat{\mathcal{C}}_\alpha$ and properly translate the encoding into the encoding for $\hat{\mathcal{C}}_\alpha$.

The demux gadget routes GC label inputs to the active branch. The demux is implemented as a garbled gadget that properly translates the encoding of the input. Now, since E holds a GC for the UC $\mathcal{U}$ (programmed as $\mathcal{C}_\alpha$) and holds inputs X_γ, she can evaluate. As Base is correct, this yields the appropriate output labels Y_γ. Finally, the mux properly translates the output; this translation table can be correctly constructed thanks to the sequential composability of Base. Therefore, GCWise is correct. □

Theorem 2. *If* Base *is oblivious and sequentially composable then* GCWise *is oblivious.*

Proof. By construction of a simulator $\mathcal{S}_{\text{obv}}$.

The goal of the simulator is to produce a tuple $(\mathcal{C}, \hat{\mathcal{C}}', X')$ such that:

$$(\mathcal{C}, \hat{\mathcal{C}}', X') \stackrel{c}{=} (\mathcal{C}, \hat{\mathcal{C}}, X)$$

where $\hat{\mathcal{C}}$ and X arise in the real world execution.

Our simulator uses Base's obliviousness simulator as a black box. There is one crucial detail in this use: we have carefully ensured that there is only one universal topology $\mathcal{U}$. Hence, the call to $\mathsf{Base}.\mathcal{S}_{\text{obv}}(1^\kappa, \mathcal{U})$ indistinguishably simulates *any* of the conditional branches.

Our definition of $\mathcal{S}_{\text{obv}}$ closely matches the definition of Ev (Figure 2). Specifically, $\mathcal{S}_{\text{obv}}$ proceeds as follows:

- Simulate the input string X by drawing uniform bits. This is trivially indistinguishable from real, since our input encoding string e is also chosen uniformly.
- Parse X as $(\hat{\alpha}, X')$.
- Simulate the bucket table and its garbled material $\hat{\mathcal{C}}_{\text{bt}}$ by calling a modular simulator $\mathsf{Sim}_{\text{bt}}(\hat{\alpha})$ (described later). Let $(\beta, \gamma, \hat{\gamma}, (B_\beta \setminus \gamma), S_{i \in [m] \setminus \gamma}, K_\beta)$ be the simulated output.

- Simulate each stack of material $\mathcal{M}_{j\neq\beta}$ by a uniform string. This is indistinguishable from real: E obtains the decryption key K_β, but does not obtain any decryption key $K_{j\neq\beta}$, so in the real world she cannot decrypt. Simulating the active bucket is more nuanced.
- Simulate the demultiplexer and its garbled material $\hat{\mathcal{C}}_{\mathsf{demux}}$ via a modular simulator $\mathsf{Sim}_{\mathsf{demux}}(\hat{\gamma}, X')$ (described later). Let X_γ be the simulated output.
- Proceed by garbling each of the (simulated) $m-1$ siblings as described in Ev. Stack each material into $\mathcal{M}_\beta$.
- Simulate the material for the active instance by calling Base's obliviousness simulator: $\hat{\mathcal{C}}_\alpha \leftarrow \mathsf{Base}.\mathcal{S}_{\mathrm{obv}}(1^\kappa, \mathcal{U})$. Stack $\hat{\mathcal{C}}$ into $\mathcal{M}_\beta$ to complete the simulation of $\mathcal{M}_\beta$. We argue indistinguishability shortly.
- Evaluate the active instance normally: $Y_\gamma \leftarrow \mathsf{Base}.\mathsf{Ev}(\mathcal{U}, \hat{\mathcal{C}}_\alpha, X_\gamma)$.
- Simulate the multiplexer and its garbled material $\hat{\mathcal{C}}_{\mathsf{mux}}$ via a modular simulator $\mathsf{Sim}_{\mathsf{demux}}(\hat{\gamma}, Y_\gamma)$ (described later).
- Output all simulated GC material.

First, note that the simulated stacked material for the active bucket $\mathcal{M}_\beta$ is indistinguishable from real. This is because (1) the materials for the $m-1$ siblings are generated by garbling, which matches the real world and hence are clearly indistinguishable, and (2) the material for the active instance $\hat{\mathcal{C}}_\alpha$ is generated by Base's obliviousness simulator. By assumption, Base is oblivious, so this additional simulated material is indistinguishable from real.

Now, the above simulation refers to three modular simulators for our GC gadgets: $\mathsf{Sim}_{\mathrm{bt}}$, $\mathsf{Sim}_{\mathsf{demux}}$, and $\mathsf{Sim}_{\mathsf{mux}}$. Each of these gadgets are implemented from typical GC techniques: namely, encrypting output values by masking the output value with a PRF applied to the correct input value. These techniques are simple and well known, so we do not fully flesh out these component simulators. However, there are two important points which we must address.

Simulation of information revealed by the bucket table. The bucket table gadget reveals information in cleartext to E: E sees the active bucket ID β and the active instance id γ. These values must be simulated.

We argue that $\mathsf{Sim}_{\mathrm{bt}}$ (1) can simulate β by uniformly sampling a value from $[\ell]$ and (2) can simulate γ by uniformly sampling a value from $[m]$. This simulation is valid because in the real world (1) we sample each offset value δ_i uniformly at random, and (2) we uniformly choose the active instance from the set of all candidate instances (see discussion in Section 5.1). This means that a given branch ID is equally likely to reside in each bucket. Moreover, we sample among each of these instances uniformly, so each bucket is equally likely to be the active bucket. Hence uniformly sampling β and γ is a good simulation.

Security of using a PRF on labels from Base. Our multiplexer gadget (Section 5.2) takes as input output labels from the underlying scheme Base. Our multiplexer is a typical gadget that encrypts garbled rows using a PRF. Hence, we must be careful: we use output labels from Base as PRF keys. To simulate, the PRF definition requires PRF keys to be chosen uniformly. Here is where we make use of sequential composability (Definition 7). Sequential composability

insists that all output labels, even jointly, are uniformly random. Thus, we can use the output labels as PRF keys without breaking the security of the PRF.

GCWise is oblivious. □

Theorem 3. *If* Base *is oblivious and sequentially composable then* GCWise *is private.*

Proof. By construction of a simulator $\mathcal{S}_{\text{prv}}$.

By Theorem 2, GCWise is oblivious, so there exists an obliviousness simulator $\mathcal{S}_{\text{obv}}$. $\mathcal{S}_{\text{prv}}$ first runs $\mathcal{S}_{\text{obv}}(1^\kappa, \mathcal{C})$ and obtains $(\mathcal{C}, \hat{\mathcal{C}}', X')$. From here, $\mathcal{S}_{\text{prv}}$ must simulate an output decoding string d' such that

$$(\hat{\mathcal{C}}, X, d) \stackrel{c}{=} (\hat{\mathcal{C}}', X', d')$$

$\mathcal{S}_{\text{prv}}$ computes $Y' \leftarrow \mathsf{Ev}(\mathcal{C}, M', X', t)$. Now, $\mathcal{S}_{\text{prv}}$ constructs d' in a straightforward manner: for each wire y, $\mathcal{S}_{\text{prv}}$ fills one of the two labels in d' with Y' at position y such that decoding the label results in cleartext output y. The other label is set to be uniform with the restriction that its least significant bit differs from Y'. This simulation is indistinguishable from the real execution. The simulated d' decodes the true output y and is indistinguishable from d. □

Theorem 4. *If the underlying garbling scheme* Base *is oblivious and sequentially composable then* GCWise *is authentic.*

Proof. Authenticity (Figure 4) demands that an adversary $\mathcal{A}$ with only $\hat{\mathcal{C}}$ and X cannot construct a garbled output Y' that is different from the one allowed by X and $\hat{\mathcal{C}}$, i.e., where $Y' \neq \mathsf{Ev}(\hat{\mathcal{C}}, X)$ and $\mathsf{De}(d, Y) \neq \perp$, except with negligible probability.

Our authenticity proof is like existing GC proofs, e.g., [ZRE15].

Authenticity follows from the definition of the privacy simulator $\mathcal{S}_{\text{prv}}$, from our choice of output decoding string d, and from De. Assume, to reach a contradiction, that a polytime $\mathcal{A}$ can indeed forge a proof. We demonstrate that such an adversary allows a privacy distinguisher. Specifically, on input $(\hat{\mathcal{C}}, X, d)$ the distinguisher (1) evaluates the GC normally to obtain Y, (2) forges an output Y' by invoking $\mathcal{A}$, and (3) outputs 1 if and only if $Y \neq Y'$ and both Y and Y' successfully decode.

If we give to this distinguisher a circuit garbling produced by $\mathcal{S}_{\text{prv}}$, the distinguisher will output one with negliglible probability. Indeed, $\mathcal{A}$ must guess $Y' \neq Y$ that successfully decodes. However, for each bit in d, $\mathcal{S}_{\text{prv}}$ uniformly samples the inactive decoding string. Thus $\mathcal{A}$ must simply *guess* such a value, since these uniformly drawn values are independent of the adversary's view. This only succeeds with probability $\frac{1}{2^\kappa}$.

Hence, if the $\mathcal{A}$ can succeed on a *real* garbling with non-negligible probability, then we indeed have distinguisher. But GCWise is private, so the distinguisher should not exist, and we have a contradiction.

GCWise is authentic. □

Acknowledgements. This work was supported in part by NSF award #1909769, by a Facebook research award, a Cisco research award, and 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. This work is also supported by DARPA under Cooperative Agreement HR0011-20-2-0025, NSF grant CNS-2001096, CNS-1764025, CNS-1718074, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Research Award and Sunday Group. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.

References

[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.

[BNO19] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 530–549. Springer, Heidelberg, June 2019.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 41–50. IEEE Computer Society, 1995.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Heidelberg, May 2020.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[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.

[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.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 536–553. Springer, Heidelberg, August 2013.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg, August 2013.

[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.

[HK21] David Heath and Vladimir Kolesnikov. Logstack: Stacked garbling with $O(b \log b)$ computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[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.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.

[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.

[KS08a] 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.

[KS08b] 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.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[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. Cryptology ePrint Archive, Report 2020/161, 2020. `https://eprint.iacr.org/2020/161`.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, page 129–139, New York, NY, USA, 1999. Association for Computing Machinery.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. LNCS, pages 94–124. Springer, Heidelberg, 2021.

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

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[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.