Garbled Circuit Lookup Tables with Logarithmic Number of Ciphertexts[*]

David Heath[1], Vladimir Kolesnikov[2], and Lucien K. L. Ng[3]

[1]daheath@illinois.edu University of Illinois Urbana-Champaign
[2]kolesnikov@gatech.edu Georgia Institute of Technology
[3]kng68@gatech.edu Georgia Institute of Technology

Abstract

Garbled Circuit (GC) is a basic technique for practical secure computation. GC handles Boolean circuits; it consumes significant network bandwidth to transmit encoded gate truth tables, each of which scales with the computational security parameter κ. GC optimizations that reduce bandwidth consumption are valuable.

It is natural to consider a generalization of Boolean two-input one-output gates (represented by 4-row one-column lookup tables, LUTs) to arbitrary N-row m-column LUTs. Known techniques for this do not scale, with naïve size-$O(Nm\kappa)$ garbled LUT being the most practical approach in many scenarios.

Our novel garbling scheme – logrow – implements GC LUTs while sending only a *logarithmic* in N number of ciphertexts! Specifically, let $n = \lceil \log_2 N \rceil$. We allow the GC parties to evaluate a LUT for $(n-1)\kappa + nm\kappa + Nm$ bits of communication. logrow is compatible with modern GC advances, e.g. half gates and free XOR.

Our work improves state-of-the-art GC handling of several interesting applications, such as privacy-preserving machine learning, floating-point arithmetic, and DFA evaluation.

Keywords: Multiparty Computation, Garbled Circuits, Lookup Tables

[*] Abridged version to appear in proceedings of IACR Eurocrypt 2024.

Contents

1 Introduction

Garbled Circuit (GC) allows two parties to jointly evaluate circuits without leaking anything that cannot be inferred from the output of the computation. In contrast with other secure computation (MPC) techniques, *e.g.* GMW, GC requires only a constant number of communication rounds, independent of the circuit's size and depth.

In GC, the garbler G steps through the circuit gate by gate. At each gate, G constructs and sends to the evaluator E an encryption (garbling) of the gate's truth table. Altogether, these messages consume significant communication. Indeed, communication is the bottleneck of GC performance, so reducing communication is a central goal of GC research.

Most prior GC communication improvements have come in the form of new methods for handling fan-in two gates. While Yao's original construction [40] required four ciphertexts per gate, subsequent works improved cost to zero ciphertexts per XOR gate and 1.5 ciphertexts per AND gate [31, 25, 33, 24, 41, 37]. Despite intense interest, progress has slowed and lower bounds began to emerge [41]. [37]'s intricate technique improves over the previous best [41] by less than 25%.

Seeking significant GC cost improvement, we look at a broader problem: efficient evaluation of general-purpose multi-input/multi-output gates. No such garbled gates were proposed (but see discussion of related techniques in Section 2). In contrast, in the non-constant-round GMW setting, multi-input/multi-output gates are known, e.g. [21, 7].

We bring multi-input/multi-output gates to GC, generalizing from fan-in-two Boolean gates to arbitrary N-row m-column *lookup tables* (LUTs).

LUT Applications. In many settings, LUTs are powerful. For instance, SiRnn [36] uses LUTs to accelerate and better approximate non-linear functions needed for machine learning, *e.g.*, $\mathsf{sigmoid}$, tanh, and $1/\sqrt{x}$. For instance, SiRnn uses LUTs of size 1020 as part of its implementation of inverse square root. LUTs can also simplify floating point arithmetic [35, 34]. [35] uses LUTs of size 128 as part of their implementation of tan and LUTs of size 256 to convert integers to floating points. [34] uses LUTs of size 2^{12} for $\mathsf{sigmoid}$ and tanh over 16-bit floating points. It is also possible to, for instance, use LUTs to implement state transition tables for Deterministic Finite Automata (DFAs), which is useful in a variety of problems, such as substring matching and DNA pattern search [38].

More generally, LUTs open opportunities for more efficient secure computation, similar to how LUTs enable more efficient plaintext computation in Field Programmable Gate Arrays (FPGA). Yet, the above applications are now only implemented with (GMW-style) interactive primitives [23, 21, 7], meaning that performance is highly susceptible to network latency.

Until our work the most concretely efficient non-interactive and composable technique for natural LUT applications (e.g. above) was the straightforward transmission of a full encrypted truth table. This consumes $\Theta(Nm\kappa)$ bits of communication, linear in the size of the table. The recent one-hot garbling technique [13] enables highly efficient GC LUTs that consume only $O(\log N \cdot \kappa)$ bits, but *only* in a setting where the evaluator E is allowed to learn which row is looked up from each table. While [13] used these *privacy-free* GC LUTs in securely computing highly structured functions with convenient algebraic properties, their technique does not apply in the less structured settings we consider.

1.1 Contribution

We realize N-row m-column GC LUTs while sending only a *logarithmic* in N number of ciphertexts per LUT! Namely, our approach securely executes programs that arbitrarily compose any number of LUTs, each potentially computing a different function, all in constant rounds.

Our construction is lean: for computational security parameter κ and for $n = \lceil \log_2 N \rceil$ our garbled LUT gate uses precisely $(n-1)\kappa + nm\kappa + Nm$ bits of communication. Our computational cost is $O((N(1+\frac{m}{\kappa}) + nm)c_\kappa + Nm\kappa)$, requiring $O(N(1+\frac{m}{\kappa}) + nm)$ evaluations of a hash function (c_κ denotes the computational

Approach	**Communication (bits)**	**Computation (bit ops.)**	**E knows f**	**E knows x**	**Amortized**
Ours	$(n-1)\kappa + nm\kappa + Nm$	$O((N(1+\frac{m}{\kappa}) + nm)c_\kappa + Nm\kappa)$			
Linear Scan	$O(Nm\kappa)$	$O(Nmc_\kappa)$			
One-Hot	$(n-1)\kappa$	$O(Nm\kappa + Nc_\kappa)$	✗	✗	
SGC	$O(n^2\kappa + nm\kappa)$	$O(N^{2.389}mc_\kappa)$	✗		
GPIR	$\tilde{O}(\sqrt{N}m\kappa)$	$\tilde{O}(Nmc_\kappa)$	✗		
GRAM	$O^!(nm\kappa + n^3\kappa)$	$O^!(nmc_\kappa + n^3c_\kappa)$			✗

Table 1: Comparison of various GC techniques for computing $[\![a]\!] \mapsto [\![f(a)]\!]$ inside GC where $f : \{0,1\}^n \rightarrow \{0,1\}^m$ is a table with $N = 2^n$ rows. Each ✗ symbol indicates a weakness of the respective approach. $\tilde{O}$ includes polylog(N) factors; $O^!$ includes polylog(n) factors. All schemes other than linear scan and GRAM require that G knows f. We emphasize the low constants in our communication.

cost of evaluating the hash function; see Section 3.1 for the hash function definition). Notably, this is a concrete and asymptotic computation improvement over linear scan of the LUT (i.e., including each of the N rows in a garbled table).

At a very high level, our protocol uses the recent One-Hot Garbling (OHG) technique [13] to evaluate a *masked* LUT on a masked index. The masks are then efficiently taken off inside GC. Efficient masking and unmasking are achieved by our new core technique, which allows the parties to efficiently evaluate a *random function* inside GC. Our technique hides the random function from the GC evaluator and only requires that the garbler send a logarithmic number of ciphertexts.

We formalize our construction as a garbling scheme [3] and prove security. Our scheme is compatible with Free XOR [25] and with Free-XOR based GC improvements, e.g. [41, 37, 13, 15]. As a garbling scheme, our construction immediately implies 2PC protocols in semi-honest, malicious, covert, and publicly verifiable covert models.

Our garbling scheme is compatible with the Stacked Garbling technique [14]; it can be used in the context of conditional branches that are "stacked".

2 Related Work

We propose a technique for securely evaluating functions inside GC. Given a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ and garbled input $[\![a]\!]$, we compute the garbled value $[\![f(a)]\!]$ (see Section 3 for detail of this notation). We evaluate f described *only* by its $N = 2^n$ row lookup table. Thus, our technique is suited to settings where f is *arbitrarily complex.*

In this review of related work, we focus on other GC approaches that can also be used to evaluate functions via their truth tables. We summarize our comparison in Table 1, and we give detailed discussion below.

Basic Boolean Gates and Linear Scans. Classic GC allows to securely compute Boolean circuits with fan-in two gates. The most recent gate-by-gate construction requires $\approx 1.5\kappa$ bits of communication per AND gate [37]; XOR gates are communication-free [25].

Boolean gates can implement lookup tables via *linear scans.* To implement a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$, the parties compute gates that touch each entry of the function's truth table. The truth table is a string of length $2^n m = Nm$, so in total $O(Nm\kappa)$ bits of communication are needed.

While expensive, simply enumerating the truth table remained the best approach to LUTs for many values N, due to the excellent constants involved in this basic technique.

Our technique asymptotically improves over basic linear scans by factor κ and is concretely superior for all values of $N \geq 4$ (see Section 6).

One-Hot Garbling (OHG) [13] enables communication-efficient *privacy-free* LUTs. The technique allows the parties to compute $[\![f(x)]\!]$ while using only $(n-1)\kappa$ bits of communication (and is reduced to $(n-1)\kappa$ bits recently [11]), but only when E knows both f and x. [13] demonstrates that this building block can be used to implement privacy-preserving computations, but only for certain functions with friendly algebraic properties, such as linearity. The technique improves highly structured functions, including binary vector outer products, integer multiplication, and more. OHG is poorly suited to *general* functions f, since arbitrary functions do not have exploitable algebraic properties.

We build on top of OHG to implement $[\![f(x)]\!]$ for arbitrary f. We require additional communication, but remove the requirement that E knows f and x.

Stacked Garbling (or Stacked GC, SGC) [22, 14, 12] is a GC improvement that allows for efficient handling of programs with conditional branching. The technique allows the garbler to send a GC proportional only to one program execution path, not to the full computation.

SGC can implement a function f by representing f as a conditional:

$$f(x) = \begin{cases} f(0) & \text{if } x = 0 \\ ... & \\ f(N-1) & \text{if } x = N-1 \end{cases}$$

This requires care. SGC uses extra gadgets to enter/exit conditionals, and these gadgets require communication. For a conditional with n input wires, m output wires, and b branches, the parties consume $O(b^2(n+m)\kappa)$ bits of communication. Thus, using SGC to implement a switch statement over the $b = N$ branches of f leads to cost $O(Nm\kappa)$, no better than a linear scan. Indeed, evaluating a large number of very small branches is not SGC's intended application [12].

A more effective approach is to use SGC *recursively*, encoding the function as a binary tree of nested IF statements. Indeed, if G and E agree on f, they can securely compute $[\![f(x)]\!]$ this way while using only $O(n^2\kappa + nm\kappa)$ bits of communication. Each recursive call to SGC requires encoding/decoding gadgets of size (only) $O((n+m)\kappa)$. This is because we only need to "pass through" $O(n)$ choice bits to the lower levels of recursion. In total, n SGC gadgets are executed, resulting in overall quadratic in n complexity. This communication performance is surprisingly good for an unintended application of SGC.

The problem is computation. SGC achieves communication improvement at the cost of computation, and in this case the increase is significant. Nested execution of SGC over N branches consumes unacceptably high computation, scaling with $O(N^{2.389}m\kappa)$ [10]. We emphasize the limits of scaling using this technique. Consider a size $N = 2^{16}$ LUT, a case suitable for 16-bit operations and easily handled by our approach. Here, SGC will require a clearly unacceptable $> 2^{38}$ CCRH evaluations for $m = 1$ (approximately a day of computation on a modern laptop). This is less attractive than a simple $O(Nm\kappa)$ linear scan.

Our technique evaluates arbitrary LUT gates, while matching computation scaling of linear scans. As a bonus, our approach allows us to hide f from E.

Garbled Private Information Retrieval. [9] recently proposed a GC extension called *Garbled Private Information Retrieval* (GPIR). In GPIR, G and E agree on a public database. Then, the GC may privately and non-interactively query one index of the database and pass the result as input to subsequent GC gates. The technique is similar to PIR in the sense that G and E jointly play a server and the GC plays a client. [9] implements GPIR for a database with N entries each of size m while using only $\tilde{O}(\sqrt{N}m\kappa)$ bits of communication ($\tilde{O}$ includes concretely significant polylog(N) factors). We note that [9] does not include concrete evaluation or estimates of their cost.

In GPIR, G and E must agree on the database. [9] point out that this requirement can be relaxed by instead agreeing on an *encrypted* database and requiring the GC to decrypt the query result. While this works, it requires non-black-box evaluation of cryptographic primitives, which is extremely expensive.

Our approach allows G to secretly choose the LUT. More importantly, for many sizes of database N, our approach has superior communication. We achieve low concrete constants and avoid the need for extra polylog(N) factors.

Garbled RAM. Garbled RAM (GRAM) [29] is a powerful GC extension that enables garbling of RAM programs. The technique allows G and E to repeatedly, securely, and non-interactively access an array, and it is possible to use GRAM to implement a function LUT. Recent works [15, 32] dramatically improved GRAM. For an array of $N = 2^n$ elements each of size m, GRAM now requires only $O^!(nm\kappa + n^3\kappa)$ communication and computation per access, where $O^!$ includes polylog(n) factors.

Our approach is better suited to evaluating LUTs than GRAM in three ways.

First, GRAM's cost is *amortized* over $O(N)$ accesses. Hence, if a function f is needed $o(N)$ times in the execution of a program, then GRAM is the wrong approach. The first time a function table is used, the parties must immediately consume $\tilde{O}(Nm\kappa)$ communication (where $\tilde{O}$ includes polylog(N) factors), *significantly* worse than even a linear scan. This required amortization means that GRAM is particularly poorly applicable when using a variety of *different* LUTs f_i in a program – players would have to initialize a separate GRAM for each f_i.

Second, GRAM's constants remain relatively high. [15]'s technique is the best GRAM for small sizes, and for $m = 128$ it only begins to outperform trivial linear-scan-based GRAM at around $N = 512$. For smaller m, GRAM will perform far worse.

Third, known (non-trivial) GRAMs are *incompatible* with Stacked Garbling, and hence so is a GRAM-based LUT implementation. Resolving GRAM-SGC incompatibility requires hiding from the GC evaluator ORAM access patterns, likely requiring a costly solution.

Our technique has lean constants and can flexibly implement an arbitrary number of different functions in a single program. At $m = 8$ and $N = 512$, our approach outperforms linear scan communication by more than 30×. Finally, our garbled LUT is compatible with Stacked Garbling (SGC) in the sense that it can appear safely in an SGC conditional branch; formally, our garbling scheme is *strongly stackable* (cf. Theorem 6).

Of course, GRAM is a powerful technology – it simply is not well suited to the LUT setting. We view GRAM and our approach as complimentary technologies.

Non-GC-based LUTs. Lookup-table-based MPC has been considered outside of GC [30, 21, 7, 5] in the multi-round setting. Our work brings efficient lookup tables to the important constant-round GC-based MPC.

We stress that solutions in our compositional non-interactive setting are inherently different, and harder to achieve. Indeed, in the non-interactive setting, one party's (garbler's) actions inherently cannot depend in any way on intermediate values (even on the masked intermediate values!) of the computation.

To practically motivate our interest in GC (vs interactive) LUT, we highlight the high cost of latency. Unless the program is highly parallelizable (e.g. matrix multiplication and other operations), multiplicative depth of the program circuit would incur significant costs due to latency. For example, recent experiments reported in [39] show $\approx 1600\times$ improvement by switching from a GMW-based multi-round solution to a constant-round GC-based protocol, with the primary factor behind the speed up being network latency. Of course, this is just an example, and the costs depend on network properties and the program itself.

Interestingly, the interactive techniques of [21, 7] are similar to ours in that their cost comprises two components, one proportional to κ (a 1-out-of-N random OT) and one proportional to the size of a truth table. While costs are similar, the constructions themselves are completely different from ours, which is in the *much* harder setting.

[5] is the latest work addressing interactive lookup-tabled-based 2PC. Their total communication cost (in bits) is as follows: $(\mathrm{MT} + 4)(2^n - n - 1) + 2m$, where MT denotes the cost of preprocessing a multiplication triple. In terms of computation, the technique executes $O(2^n)$ OTs in the preprocessing phase. Using communication-efficient OT [4], the total cost is $O(2^n + m)$ bits, which does not scale with the total truth table size $2^n m$. Our communication scales with the full truth table, but our technique is non-interactive.

Private simultaneous messaging (PSM) is an MPC special case that considers several senders, each with private input, and a single referee who receives a function f of these inputs [8, 18, 19, 2]. The original PSM construction [8] showed how to evaluate arbitrary LUTs in this setting. In their construction, which works for two senders and the referee, the senders randomize (mask) the LUT and shuffle its rows. One sender then sends to the referee a message proportional to the LUT size; the other sends a pointer to a row in the LUT and a mask. This allows the referee to decrypt (only) the single selected row, obliviously yielding the function output. [2] extends this technique to generalize to multiple senders: each sender applies a random mask to the LUT and then reveals a portion of its mask, depending on its plaintext input. This similarly allows the referee to learn (only) the right LUT row.

We have a related step in our construction: We mask the LUT and then unmask portions of it based on each (encrypted) bit of the LUT's input. We approach and solve the problem in a much more complex non-interactive composable (GC) setting. [2] works with plaintext input and output, complicating composition. Further, *each* [2]'s party communicates proportionally to LUT size. It is unclear how to port the [8] and [2] approach to the non-interactive GC setting without incurring factor-κ blow-up, resulting in performance similar to classical Yao (Linear Scan in Table 1).

[41]'s Lower Bound. [41] proved a lower bound on the communication needed per GC AND gate. They define *linear* garbling schemes and show that any linear scheme *must* use at least 2κ bits per AND gate. While [37] recently circumvented this lower bound by working outside the linear definition and achieved $\approx 1.5\kappa$ bits per AND gate, the [41] lower bound still seems to imply intense difficulty in substantially improving GC gates.

Our work circumvents the [41] lower bound in two ways. First, we work with larger gates, not just two-input one-output AND gates. Second, we leverage the ability of G to send to E a cleartext truth table; this sending of a truth table is outside [41]'s definition of linearity. We believe that our work demonstrates that while basic GC Boolean gates are hard to improve, opportunities may remain to significantly improve GC overall.

3 Preliminaries

3.1 Notation and Assumptions

- κ denotes the computational security parameter (*e.g.*, 128).
- G is the GC garbler. We refer to G by he/him.
- E is the GC evaluator. We refer to E by she/her.
- We denote by $\langle\langle x, y \rangle\rangle$ a pair of values where G holds x and E holds y.
- We work with extensively with bit vectors interpreted as arrays:
  - If $a \in \{0,1\}^n$ is a vector, we denote the ith entry of a by $a[i]$.
  - We use zero-based indexing. An array $a \in \{0,1\}^n$ is the sequence of elements $(a[0], ..., a[n-1])$.
  - $a[i:j]$ denotes the sub-array of elements $(a[i], ..., a[j-1])$.
  - For an array $a \in \{0,1\}^n$, $a[0]$ is the least significant bit and $a[n-1]$ is the most significant bit.
- $[x]$ (without any prefix) denotes the set $\{0, 1, \ldots, x-1\}$.
- N is the number of rows in the lookup table.
- n is the number of bits needed to index a LUT, *i.e.*, $n = \lceil \log_2 N \rceil$.
- m is the number of the LUT's columns, equal to the number of output bits.

- $\mathcal{H}(a)$ is the one-hot vector encoding of a (cf. Definition 4).
- $\mathcal{T}(f)$ is the truth table of a function f (cf. Definition 5).
- H is a circular correlation robust hash function (CCRH, Definition 1).
- v denotes a nonce, usually an argument to H. G and E publicly agree on the value of each nonce.
- c_κ denotes the computational cost of evaluating H.

We assume a circular correlation robust hash function H [6]. We use the following definition, given by [41]:

Definition 1 (Circular Correlation Robustness)**.** *Let H be a function. We define two oracles:*

- $circ_\Delta(i, x, b) \triangleq H(x \oplus \Delta, i) \oplus b\Delta$ *where* $\Delta \in 1\{0,1\}^{\kappa-1}$.
- $\mathcal{R}(i, x, b)$ *is a random function with* κ*-bit output.*

A sequence of oracle queries (i, x, b) is legal *when the same value (x, i) is never queried with different values of b. H is circular correlation robust if for all poly-time adversaries $\mathcal{A}$:*

$$\left| \Pr_{\Delta}\left[\mathcal{A}^{circ_\Delta}(1^\kappa) = 1\right] - \Pr_{\mathcal{R}}\left[\mathcal{A}^{\mathcal{R}}(1^\kappa) = 1\right] \right| \text{ is negligible.}$$

3.2 Garbled Sharing

We build on Free XOR garbling [25]. For each bit that appears on a GC wire, we arrange that G and E hold a kind of sharing called a *garbled sharing*, a notation introduced in [13]. G's share contains a pair of length-κ *labels*. One label corresponds to a logical zero, the other to a logical one. Meanwhile, E's share contains a specific label from G's pair. Thus, the two shares together specify the value on the wire. More precisely:

Definition 2 (Garbled Sharing [13])**.** *Let $a \in \{0,1\}$ be a bit. Let $A \in \{0,1\}^\kappa$ be a bitstring. We say that the pair $\langle\!\langle A, A \oplus a\Delta \rangle\!\rangle$ is a* garbled sharing *of a over (usually implicit) $\Delta \in 1\{0,1\}^{\kappa-1}$. I.e., Δ is uniform except that its least significant bit is 1. We denote a garbled sharing of a by writing $[\![a]\!]$:*

$$[\![a]\!] \triangleq \langle\!\langle A, A \oplus a\Delta \rangle\!\rangle$$

Note that in our definition, the zero label is the bitstring A, whereas the one label is the bitstring $A \oplus \Delta$ where Δ is global to the circuit. As defined and used in this work, garbled sharing $[\![a]\!]$ is Free XOR-specific.

Garbled Arrays. We generalize from sharings of bits to sharings of arrays. Let $a \in \{0,1\}^n$ be an array. We use $[\![a]\!]$ to denote the bit-by-bit encoding of a:

$$[\![a]\!] \triangleq ([\![a[0]]\!], ..., [\![a[n-1]]\!])$$

Free XOR. Garbled sharings are XOR-homomorphic [25]:

$$[\![a]\!] \oplus [\![b]\!] = [\![a \oplus b]\!] \qquad \text{Definition 2}$$

Injecting G's secrets. Garbled sharings allow G to easily inject bits into the circuit. Namely, let $a \in \{0,1\}$ be a bit chosen by G. To inject his input, the parties simply use the following pair:

$$\langle\!\langle a\Delta, 0 \rangle\!\rangle = [\![a]\!] \qquad \text{Definition 2}$$

This capability is used not only for G's top-level input, but also to provide auxiliary bits needed for our construction.

Revealing bits to E. At times, it is useful for the GC to reveal particular wire values to E (while taking care to preserve input privacy). It is easy to decrypt a particular wire value to E. Note that Definition 2 enforces that the least significant bit of Δ is a one. To reveal the cleartext value of a sharing $[\![a]\!] = \langle\!\langle A, A \oplus a\Delta \rangle\!\rangle$, G can send to E the least significant bit of his garbled share $\mathsf{lsb}(A)$. Then, E computes:

$$\mathsf{lsb}(A) \oplus \mathsf{lsb}(A \oplus a\Delta) = \mathsf{lsb}(A) \oplus \mathsf{lsb}(A) \oplus a \cdot \mathsf{lsb}(\Delta) = a$$

Revealing a bit to E requires only one bit of communication.

3.3 Garbling Schemes

We formalize our approach as a *garbling scheme* [3].

Definition 3 (Garbling Scheme [3])**.** *A* garbling scheme *is a tuple of algorithms (Gb, Ev, En, De) that specify how to garble/evaluate a circuit:*

- *$Gb(1^\kappa, \mathcal{C}) \to (\mathcal{M}, e, d)$ garbles circuits. It takes as input the security parameter κ and a circuit description $\mathcal{C}$. The procedure outputs garbled circuit material $\mathcal{M}$ as well as two strings e and d that respectively contain information needed to encode inputs and decode outputs.*

- *$En(e, x) \to X$ encodes the party inputs. It takes as input the encoding string e and a cleartext input $x \in \{0,1\}^n$ and outputs E's input wire labels X.*

- *$Ev(\mathcal{M}, X) \to Y$ evaluates GCs. It takes as input material $\mathcal{M}$ and wire labels X and outputs wire labels Y.*

- *$De(d, Y) \to y$ decodes circuit outputs. It takes as input the output decoding string d and E's output wire labels Y. The procedure outputs cleartext $y \in \{0,1\}^m$. The procedure may also output $\perp$ to indicate failure.*

A garbling scheme factors circuit evaluation $\mathcal{C}(x)$ into multiple steps. Namely, for all $\mathcal{C}$ and x, [3] insist that the following correctness condition holds:

$$De(d, Ev(\mathcal{M}, En(e, x))) = \mathcal{C}(x) \qquad\qquad \text{where } (\mathcal{M}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$$

The crucial [3] security property is *obliviousness*, which states that the pair of material and input labels $(\mathcal{M}, X)$ can be simulated. Obliviousness is the basis[1] for GC-based protocols, because it implies that an evaluator who views the GC cannot deduce anything about the garbler's input.

We formalize our garbling scheme in Section 5. We provide definitions for garbling scheme properties and prove that our scheme satisfies them in Section 7.

Projectivity. [3]'s framework allows for a variety of schemes that support non-Boolean encoded values, but our scheme only handles Boolean wires. Formally, our scheme is *projective* [3], which means that each circuit wire is associated with two labels that respectively encode logical 0/logical 1. Projective schemes have standard and simple definitions for En and De.

From schemes to protocols. Garbling schemes have been used as the basis for protocols in the semi-honest, malicious, covert, and PVC models [3, 26, 17, 28, 16]. Hence, our construction implies 2PC protocols in these models.

- Parameters: Parties agree on input size n
- Input:
  - Parties input a sharing $\llbracket x \rrbracket$ where $x \in \{0,1\}^n$.
  - E inputs x.
- Output:
  - Parties output a sharing $\langle\langle X_G, X_E \rangle\rangle = \llbracket \mathcal{H}(x) \rrbracket$ such that for each index i:

$$X_E[i] = \begin{cases} X_G[i] & \text{if } i \neq x \\ X_G[i] \oplus \Delta & \text{otherwise} \end{cases}$$

- Communication: G sends to E $(2n-1)\kappa$ bits.
- Computation: Each party uses $O(2^n c_\kappa)$ computation.

Figure 1: The interface to the key One-Hot Garbling [13] operation. The operation enables the parties to efficiently compute $\llbracket x \rrbracket \mapsto \llbracket \mathcal{H}(x) \rrbracket$. Note that E must know x in the clear. Our construction builds on this operation.

3.4 One-Hot Garbling

One-Hot Garbling [13] is a recent GC improvement that goes beyond Boolean gate evaluation. The technique allows *privacy-free* (i.e. with E learning the accessed index) GC LUTs where communication is logarithmic in the number of LUT rows N. The technique reduces communication consumption via so-called *garbled one-hot encodings.* We build on top of one-hot garbling to achieve efficient *privacy-preserving* GC LUT.

[13] Review. It is useful to construct *one-hot* encodings inside the GC:

Definition 4 (One-Hot Encoding)**.** *Let $x \in \{0,1\}^n$ be a bitstring. The* one-hot encoding *of x is a length-2^n bitstring denoted $\mathcal{H}(x)$ that is zero everywhere, except at index x, where it is one:*

$$\mathcal{H}(x)[i] \triangleq \begin{cases} 1 & \textit{if } i = x \\ 0 & \textit{otherwise} \end{cases}$$

Consider the case where the parties hold an n-bit share $\llbracket x \rrbracket$ and where E knows x in the clear. [13]'s construction allows E to efficiently compute a length-2^n garbled vector $\llbracket \mathcal{H}(x) \rrbracket$. We list the interface to [13]'s procedure in Figure 1.

[13] leverage their new procedure in conjunction with the fact that one-hot encodings are, in a sense, fully homomorphic. Given a one-hot encoding $\mathcal{H}(x)$, we can compute $f(x)$ via *a sequence of XORs alone.* More precisely, we can multiply the one-hot vector with f's truth table:

Definition 5 (Truth Table)**.** *Let $f : \{0,1\}^n \to \{0,1\}^m$ be an arbitrary function. The* truth table *for f, denoted $\mathcal{T}(f)$, is a $2^n \times m$ matrix where:*

$$\mathcal{T}(f)[i][j] = f(i)[j]$$

[13] exploits the following simple but crucial fact:

[1] [3] also defines *privacy* and *authenticity.* While these security properties are technically incomparable with obliviousness, for most GC schemes (including ours) they follow easily from obliviousness. We prove that our scheme satisfies all three properties in Section 7.

Lemma 1. *Let $f : \{0,1\}^n \to \{0,1\}^m$ be an arbitrary function and let $x \in \{0,1\}^n$ be a bitstring:*

$$\mathcal{T}(f)^\intercal \cdot \mathcal{H}(x) = f(x)$$

Proof. Intuitively, the one-hot encoding $\mathcal{H}(x)$ "selects" row x of the truth table.

More precisely, consider the i-th bit of $\mathcal{T}(f)^\intercal \cdot \mathcal{H}(x)$:

$$\begin{aligned}
&(\mathcal{T}(f)^\intercal \cdot \mathcal{H}(x))[i] \\
= &\bigoplus\nolimits_j \mathcal{T}(f)^\intercal[i][j] \cdot \mathcal{H}(x)[j] && \text{Definition Matrix Mult.} \\
= &\bigoplus\nolimits_j \mathcal{T}(f)[j][i] \cdot \mathcal{H}(x)[j] && \text{Definition Matrix Transpose} \\
= &\,\mathcal{T}(f)[x][i] && \text{Definition 4} \\
= &\,f(x)[i] && \text{Definition 5}
\end{aligned}$$

Since this holds for each output bit i, we have $\mathcal{T}(f)^\intercal \cdot \mathcal{H}(x) = f(x)$ □

The upshot of Lemma 1 is that if the parties hold $[\![x]\!]$ and if E knows x, then the parties can compute $[\![\mathcal{H}(x)]\!]$ via Figure 1, then compute:

$$\mathcal{T}(f)^\intercal \cdot [\![\mathcal{H}(x)]\!] = [\![f(x)]\!]$$

This matrix multiplication is well-defined thanks to Free XOR (see Section 3.2). In other words, if the parties can afford to write out the long encoding $[\![\mathcal{H}(x)]\!]$, then they can efficiently compute $[\![f(x)]\!]$ for *any* function f and without using any additional communication. However, the technique only works if E knows the one-hot active location x.

We build on top of basic one-hot garbling to evaluate arbitrary functions f even when E does not know the function's argument and even when she does not know f.

One bit output special case. In Section 4, we introduce our technique at a high level. There, we for simplicity specialize to functions with only one bit of output. This allows us to consider the following corollary of Lemma 1:

Corollary 1. *Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary function and $x \in \{0,1\}^n$ be a bitstring:*

$$\langle \mathcal{T}(f) \cdot \mathcal{H}(x) \rangle = f(x)$$

Above, $\langle x \cdot y \rangle$ denotes the inner product of vectors x and y. By Corollary 1 and Free XOR, the following holds for any $f : \{0,1\}^n \to \{0,1\}$:

$$\langle \mathcal{T}(f) \cdot [\![\mathcal{H}(x)]\!] \rangle = [\![f(x)]\!]$$

In our formal construction (see Section 5), we consider the general case of functions with m bits of output.

4 Technical Overview

In this section, we present a detailed overview of our construction. We proceed in two steps. First, Section 4.1 demonstrates a reduction from securely computing an arbitrary function f to computing a *random* function r. Evaluating random functions often helps with evaluation of general functions (e.g., [1, 8, 20] and many more). Second, Section 4.2 introduces our **main contribution** – an efficient procedure for evaluating a random function inside GC.

For simplicity, in this section we introduce our construction for a function $f : \{0,1\}^n \to \{0,1\}$ with only one bit of output. Our formal construction (see Section 5) generalizes to LUTs with m bits of output.

4.1 Reducing Lookup Tables to Random Function Evaluation

Our starting point is the one-hot garbling technique [13] (see Section 3.4). Recall that if the parties hold $[\![a]\!]$ and if E knows a, then Figure 1 can be leveraged to evaluate an arbitrary function f:

$$\langle \mathcal{T}(f) \cdot [\![\mathcal{H}(a)]\!] \rangle = [\![f(a)]\!]$$

Thus, one-hot garbling directly enables *privacy-free LUTs*. Our scheme builds on top of these privacy-free LUTs to implement *privacy-preserving* garbled LUTs. Namely, the looked up index remains hidden from E and, as a bonus, the content of the table can be chosen by G and hidden from E. To achieve this, we carefully introduce a mask on the index and a mask on the table itself. The mask on the table is generated from seeds in a way that uses low communication.

Masking a. Privacy-free LUTs leak the evaluation point a to E. As a starting point, we add a uniform mask α to a. (In fact, we use the least significant bit of G's share of $[\![a]\!]$ as α; this allows us to cleanly introduce a mask without sending extra bits). Rather than computing $\mathcal{H}(a)$, we instead compute $\mathcal{H}(a \oplus \alpha)$ where α is a uniform mask. However, there remains a problem in the *usage* of this vector. We cannot directly multiply $\mathcal{H}(a \oplus \alpha)$ by a truth table, because our evaluation point a is no longer the distinguished location of the one-hot vector. As a first attempt, the parties could try agreeing on the following function:

$$\text{bad}(x) \triangleq f(x \oplus \alpha)$$

Then, the parties could compute:

$$\langle \mathcal{T}(\text{bad}) \cdot [\![\mathcal{H}(a \oplus \alpha)]\!] \rangle = [\![\text{bad}(a \oplus \alpha)]\!] = [\![f(a)]\!]$$

While correct, this is insecure. To compute her share, E must *know* bad, and this leaks α, and thus a.

Masking the LUT. Suppose we can efficiently compute inside the GC a *uniformly random* function $r : \{0,1\}^n \to \{0,1\}$ whose value is known to G but unknown to E; we will show how to do this shortly. We define a new function f':

$$f'(x) \triangleq f(x \oplus \alpha) \oplus r(x)$$

Now, G can safely send $\mathcal{T}(f')$ to E: this table leaks nothing about f because the table is masked by r. Note, this transmission scales linearly with $N = 2^n$, but is *independent* of the computational security parameter κ: G sends E the full *cleartext* truth table.

The GC now conveys to E the cleartext value $x \triangleq a \oplus \alpha$. Again, suppose for now that the parties can somehow compute $[\![r(x)]\!]$; we discuss this core contribution in Section 4.2. The parties compute:

$$\begin{aligned}
&\langle \mathcal{T}(f') \cdot [\![\mathcal{H}(x)]\!] \rangle \oplus [\![r(x)]\!] && \\
&= \langle \mathcal{T}(f') \cdot [\![\mathcal{H}(a \oplus \alpha)]\!] \rangle \oplus [\![r(a \oplus \alpha)]\!] && \text{Definition } x \\
&= [\![f'(a \oplus \alpha)]\!] \oplus [\![r(a \oplus \alpha)]\!] && \text{Corollary 1} \\
&= [\![f((a \oplus \alpha) \oplus \alpha) \oplus r(a \oplus \alpha)]\!] \oplus [\![r(a \oplus \alpha)]\!] && \text{Definition } f' \\
&= [\![f(a)]\!] &&
\end{aligned}$$

This computation hides a. E observes the point $a \oplus \alpha$, but α masks a. Moreover, the function r masks the truth table for f and ensures that E cannot use the truth table to deduce a.

Thus, if we can securely evaluate $[\![r(x)]\!]$, then we can securely evaluate $[\![f(a)]\!]$.

4.2 Evaluating a Uniformly Random Function $[\![r(x)]\!]$

Suppose that the parties hold both $[\![x]\!]$ and $[\![\mathcal{H}(x)]\!]$; the computation of one-hot encoding $[\![\mathcal{H}(x)]\!]$ was already discussed in Section 4.1. Our goal is to build a procedure that cheaply implements the following:

$$[\![x]\!], [\![\mathcal{H}(x)]\!] \mapsto [\![r(x)]\!]$$

where $r : \{0,1\}^n \to \{0,1\}$ is uniformly random and hidden from E.

As we will see, we will construct r from a (logarithmic in N) number of "half-hidden" uniform functions.

Half-hidden uniform functions. Our crucial insight is that it is possible to efficiently evaluate a uniformly random function $[\![r_0(x)]\!]$ where $r_0 : \{0,1\}^n \to \{0,1\}$ such that E learns only *half* of the truth table for r_0. We later show how to use multiple such functions to account for the leaked half.

Before starting, we establish useful notation. Let $\langle\!\langle X_G, X_E\rangle\!\rangle = [\![\mathcal{H}(x)]\!]$ be G and E's shares of the garbled one-hot vector. Define r_0's truth table $R_0 \triangleq \mathcal{T}(r_0)$. We will be working extensively with the left and right halves of vectors, so for convenience, we set the following (recall, $A[i : j] = (A[i], ..., A[j-1])$):

$$\begin{aligned} R_0^\ell &\triangleq R_0[0 : N/2] \qquad & R_0^r &\triangleq R_0[N/2 : N] \\ X_G^\ell &\triangleq X_G[0 : N/2] \qquad & X_G^r &\triangleq X_G[N/2 : N] \\ X_E^\ell &\triangleq X_E[0 : N/2] \qquad & X_E^r &\triangleq X_E[N/2 : N] \\ \mathcal{H}(x)^\ell &\triangleq \mathcal{H}(x)[0 : N/2] \qquad & \mathcal{H}(x)^r &\triangleq \mathcal{H}(x)[N/2 : N] \end{aligned}$$

Now, consider $[\![x[n-1]]\!]$, the most significant bit of $[\![x]\!]$, and recall that E's share of this bit is one of two possible labels: Y or $Y \oplus \Delta$, where Y is G's share of the bit. G *defines* the function r_0 by applying a hash function H to each of these labels (appropriately setting length of H's output):

$$R_0^\ell \triangleq H(v_R, Y) \qquad R_0^r \triangleq H(v_R, Y \oplus \Delta)$$

Here, v_R is a fresh nonce. If $x[n-1] = 0$, then E holds Y, so she can locally compute the left half of the truth table $H(v_R, Y) = R_0^\ell$; else she can compute the right half R_0^r. For sake of example, suppose that $x[n-1]$ is zero, so E learns the left half of the table; the one case is symmetric.

Recall that if E knew the *entire* truth table for r_0, then since the parties hold $[\![\mathcal{H}(x)]\!]$, they could compute:

$$\langle R_0 \cdot [\![\mathcal{H}(x)]\!]\rangle = \langle \mathcal{T}(r_0) \cdot [\![\mathcal{H}(x)]\!]\rangle = [\![r_0(x)]\!]$$

However, as E only knows the first half of the table, she can only compute (her share of the garbling of) *half* of the summands in the above inner-product:

$$\langle R_0^\ell \cdot X_E^\ell\rangle$$

E cannot directly compute the second "half" $\langle R_0^r \cdot X_E^r\rangle$, but notice that the corresponding indices of the one-hot encoding $\mathcal{H}(x)^r$ all hold zeros; this is guaranteed by the fact that the single one-hot active position is in the range 0 to $N/2$ when $x[n-1] = 0$. Thus in our example, $X_G^r = X_E^r$.

G does not know which half of the table E is missing, but he *does* know the encoding of zero for each index of the one-hot vector, so he can precompute both possible sums that E could be missing. He encrypts these halves such that E can decrypt only her missing half:

$$H(v_{\text{row}}, Y \oplus \Delta) \oplus \langle R_0^\ell \cdot X_G^\ell\rangle \oplus Z \qquad H(v_{\text{row}}, Y) \oplus \langle R_0^r \cdot X_G^r\rangle \oplus Z$$

Above, v_{row} is a fresh nonce and $Z \in \{0,1\}^\kappa$ is a uniform string. G sends these two ciphertexts to E. (In fact, we can remove one of these two ciphertexts via the classic garbled row reduction technique [31].) In our example, E decrypts the second row and adds the result to the sum she already computed:

$$\begin{aligned} &\langle R_0^\ell \cdot X_E^\ell\rangle \oplus \langle R_0^r \cdot X_G^r\rangle \oplus Z \\ &= \langle R_0^\ell \cdot X_E^\ell\rangle \oplus \langle R_0^r \cdot X_E^r\rangle \oplus Z && x[n-1] = 0 \implies X_G^r = X_E^r \\ &= \langle R_0 \cdot X_E\rangle \oplus Z \\ &= \langle \mathcal{T}(r_0) \cdot X_E\rangle \oplus Z && \text{Definition } R_0 \end{aligned}$$

Meanwhile, G locally computes $\langle \mathcal{T}(r_0) \cdot X_G \rangle \oplus Z$, matching E's share. Thus, the parties hold:

$$\begin{aligned}
&\langle\!\langle \langle \mathcal{T}(r_0) \cdot X_G \rangle \oplus Z, \langle \mathcal{T}(r_0) \cdot X_E \rangle \oplus Z \rangle\!\rangle && \\
&= [\![\langle \mathcal{T}(r_0) \cdot \mathcal{H}(x) \rangle]\!] && \langle\!\langle X_G, X_E \rangle\!\rangle = [\![\mathcal{H}(x)]\!] \\
&= [\![r_0(x)]\!] && \text{Corollary 1}
\end{aligned}$$

Therefore, G and E can compute $[\![r_0(x)]\!]$ while leaking only half of r_0's function table to E and while consuming only κ bits of communication. The key idea was to reveal to E one half of the truth table, allowing her to apply that half via a linear map, and G accounts for the second hidden half by sending a ciphertext.

Masking the opened half via recursion. We showed how to compute $[\![r_0(x)]\!]$ where E knows the half of r_0's truth table containing index x. Recall our goal is to evaluate $[\![r(x)]\!]$ where r is *fully* hidden from E. Observe that we can construct and evaluate a new hidden uniform function $r' : \{0,1\}^{n-1} \to \{0,1\}$ and define

$$r(x) \triangleq r_0(x) \oplus r'(x[0 : n-1])$$

If r' is hidden from E, then so is r: we leaked $N/2$ bits of r_0's truth table, and the $N/2$ bits in $\mathcal{T}(r')$ (literally) cover those revealed bits. Indeed, each index of r is masked either by (1) the hidden parts of r_0 or (2) the function r'. Thus, our new task is to evaluate a secret uniform function $[\![r'(x[0 : n-1])]\!]$, which would accomplish our goal.

But this task is just a smaller version of the same problem we are already trying to solve! Indeed, we are designing a procedure to map $[\![x]\!], [\![\mathcal{H}(x)]\!] \to [\![r(x)]\!]$, and hence we can solve the problem by simply recursively computing:

$$[\![x[0 : n-1]]\!], [\![\mathcal{H}(x[0 : n-1])]\!] \mapsto [\![r'(x[0 : n-1])]\!]$$

We terminate the recursion when the parties need evaluate a uniform function with 0 bits of input; in this base case, G simply injects a uniform bit.

The needed one-hot encoding $[\![\mathcal{H}(x[0 : n-1])]\!]$ can be computed from $[\![\mathcal{H}(x)]\!]$ via simple linear operations alone. Namely, for each index of the output vector, the parties simply XOR two corresponding indices from the input vector:

$$[\![\mathcal{H}(x[0 : n-1])]\!] = [\![\mathcal{H}(x)]\!][0 : N/2] \oplus [\![\mathcal{H}(x)]\!][N/2 : N]$$

Thus, we can indeed efficiently apply our procedure recursively.

Unwinding the recursion. It is instructive to consider the direct, non-recursive definition of r. Unwinding the recursion, we see:

$$r(x) \triangleq \bigoplus_{i=0}^{n} r_i(x[0 : n-i])$$

Each function r_i is a uniform function chosen by the hash function H, and E views half of the truth table of each $r_{i<n}$. At the base case, the function r_n takes no bits of input; as output, G injects a uniform bit which is trivially hidden from E. Together, the functions r_i hide r from E; see Figure 2 for an example.

In sum, our approach allows G and E to efficiently compute a uniform function r from a sequence of half-hidden uniform functions r_i. We use r to hide the function f, which in turn allows the parties to securely compute $[\![f(a)]\!]$.

5 Approach

In this section, we formalize our approach as a *garbling scheme* (Definition 3).

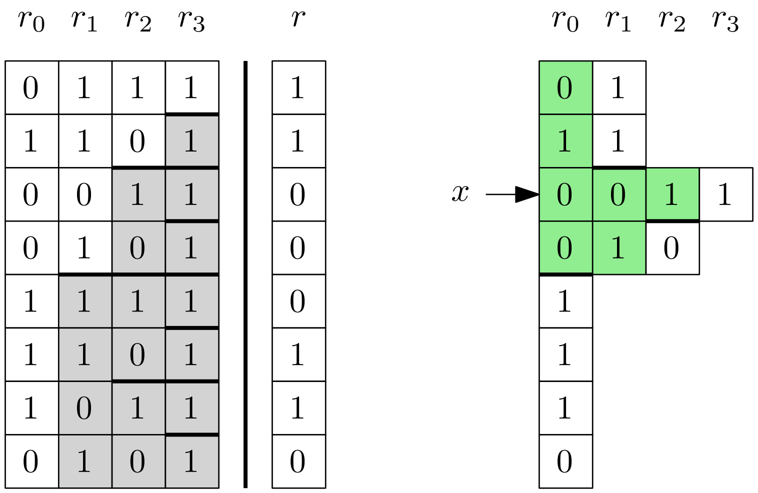

Figure 2: G recursively composes the uniform function $r : \{0,1\}^n \to \{0,1\}$ from the XOR of $n+1$ uniform functions $r_i : \{0,1\}^{n-i} \to \{0,1\}$ (see left). We depict an example where $n = 3/N = 8$. Suppose that the evaluation point x is 2. Our construction reveals to E the half of each function $r_{i<n}$ that holds index x (see right; E learns the entries depicted in green and not the entries depicted in white). r_n is a function of zero bits and so G represents the function by injecting a secret uniform bit. Ultimately, E learns nothing about r because in each row there is at least one uniform bit that is XORed in and that she does not know.

On the function output length m. For readability, we presented our technical overview for function output length $m = 1$. There are efficiency gains from batching over m bits of output rather than applying m separate LUTs. Specifically, we amortize costs associated with constructing one-hot encodings across the m output bits. Our formal construction is presented for general m. Consequently, one notable change in this section as compared to Section 4 is that we use matrix products (Lemma 1) rather than inner products (Corollary 1) to apply truth tables to one-hot encodings.

On garbled sharing notation. We present our construction using the language of garbled sharing (Definition 2), as was first done by [13]. We find this notation simple and clear, as it allows to simultaneously formally discuss garbling, evaluation, and wire value encoding/sharing. Formally, each of our figures presents *two* procedures, one executed by G and one executed by E. Our figures *never* specify that E sends a message to G. When we write that G sends a message to E, this formally means that G's procedure appends the message to the garbled circuit material and that E reads the message from the material.

Our Construction. We now formalize our garbling scheme. The most important part of our construction is specified in Figure 3 by reference to Figures 1, 4 and 5. These figures formalize G's and E's handling of LUT gates.

Aside from these figures, our formalism is relatively standard. The following construction plugs LUT gate handling into a garbling scheme that we later (in Section 7) prove satisfies [3]'s considered security notions.

Construction 1. logrow *is a* garbling scheme *(Definition 3) that supports circuits with three gate types:*

- *Standard two-input, one-output XOR gates and AND gates.*
- *LUT gates. A LUT gate is parameterized over function $f : \{0,1\}^n \mapsto \{0,1\}^m$ with LUT $\mathcal{T}(f)$. It takes as input a bitstring $a \in \{0,1\}^n$ and outputs $f(a)$.*

The garbling procedure are defined as follows:

- *$Gb(1^\kappa, \mathcal{C})$ proceeds in several steps:*

- Parameters: Parties agree on input size n and output size m.
- Input:
  - G inputs a function $f : \{0,1\}^n \to \{0,1\}^m$.
  - Parties input a shared function input $[\![a]\!]$ where $a \in \{0,1\}^n$.
- Output: Parties output a sharing $[\![f(a)]\!]$.
- Communication : G sends to E $(2n-1)\kappa + nm\kappa + 2^n m$ bits.
- Computation: Parties use $O(2^n c_\kappa + 2^n m\kappa + nmc_\kappa + 2^n mc_\kappa/\kappa)$ computation.
- Procedure:
  - Let $\langle\!\langle A, A \oplus a\Delta \rangle\!\rangle = [\![a]\!]$.
  - G computes $\alpha \triangleq \mathsf{lsb}(A)$. Let $x = a \oplus \alpha$. E computes:

$$\mathsf{lsb}(A \oplus a\Delta) = \mathsf{lsb}(A) \oplus a \cdot \mathsf{lsb}(\Delta) = \alpha \oplus a = x$$

  - G injects $\langle\!\langle \alpha\Delta, 0 \rangle\!\rangle = [\![\alpha]\!]$ as input. Parties compute $[\![x]\!] = [\![a]\!] \oplus [\![\alpha]\!]$.
  - Parties use one-hot garbling (Figure 1) to compute $[\![\mathcal{H}(x)]\!]$.
  - Parties use Figure 5 to compute $[\![r(x)]\!]$ where $r : \{0,1\}^n \to \{0,1\}^m$ is a uniformly random function. As a side-effect, G now holds r.
  - Let f' be a function that computes the following:

$$f'(x) \triangleq f(x \oplus \alpha) \oplus r(x)$$

  - G computes the truth table $\mathcal{T}(f')$ for f' and sends $\mathcal{T}(f')$ to E.
  - Parties compute and output:

$$\begin{aligned}
&(\mathcal{T}(f')^{\mathsf{T}} \cdot [\![\mathcal{H}(x)]\!]) \oplus [\![r(x)]\!] && \\
&= [\![f'(x)]\!] \oplus [\![r(x)]\!] && \text{Lemma 1} \\
&= [\![f'(a \oplus \alpha)]\!] \oplus [\![r(a \oplus \alpha)]\!] && \text{Definition } x \\
&= [\![f((a \oplus \alpha) \oplus \alpha) \oplus r(a \oplus \alpha)]\!] \oplus [\![r(a \oplus \alpha)]\!] && \text{Definition } f' \\
&= [\![f(a)]\!] &&
\end{aligned}$$

Figure 3: Garbled LUT Evaluation. We reduce the evaluation $[\![f(a)]\!]$ for arbitrary f to the evaluation $[\![r(a \oplus \alpha)]\!]$ for random r (see Figure 4).

- *Uniformly sample* $\Delta \in_\$ 1\{0,1\}^{\kappa-1}$.
- *For each circuit input* $x[i]$, *sample uniform zero label* $X[i] \in_\$ \{0,1\}^\kappa$.
- *Assemble the input encoding string* e *as a vector of pairs of labels:*

$$e[i] = (X[i], X[i] \oplus \Delta)$$

  This choice of e *is consistent with* projectivity.
- *Step through the circuit gate by gate. For each XOR gate, XOR the gate's input zero labels and place the result on the output wire [25]. For each AND gate, run the AND gate garbling procedure formalized by [41]. For each LUT gate, run G's procedure described in Figure 3. Let* $\mathcal{M}$ *be the string of material concatenated from all "messages sent to E".*

- Parameters: Parties agree on input size n and output size m.
- Input:
  - Parties input a sharing $\llbracket x \rrbracket$ where $x \in \{0,1\}^n$.
  - Parties input a shared one-hot vector $\llbracket \mathcal{H}(x) \rrbracket$.
  - E inputs x.
- Output:
  - G outputs a function $r : \{0,1\}^n \to \{0,1\}^m$ that has a uniform truth table.
  - Parties output a sharing $\llbracket r(x) \rrbracket$.
- Communication: G sends to E $nm\kappa$ bits.
- Computation: Each party uses $O(2^n m\kappa + nmc_\kappa + 2^n mc_\kappa/\kappa)$ computation.
- Procedure:
  - If $n = 0$:
    - G samples uniform $s \in_\$ \{0,1\}^m$ as the output of $r : \emptyset \to \{0,1\}^m$.
    - Parties output $\llbracket r(x) \rrbracket = \langle\!\langle s\Delta, 0 \rangle\!\rangle$ and halt.
  - Parties evaluate a half-hidden uniform function $\llbracket \hat{r}(x) \rrbracket$ (Figure 5), where $\hat{r} \in \{0,1\}^n \to \{0,1\}^m$. G learns $\hat{r}$ during evaluation. As a side effect, E learns "half" of $\hat{r}$.
  - Parties evaluate another uniform function $\llbracket r'(x[0:n-1]) \rrbracket$ by recursively invoking this procedure, where $r' \in \{0,1\}^{n-1} \to \{0,1\}^m$, with the following specification:
    - Parties agree on input size $n-1$ and output size m.
    - Parties input $\llbracket x[0:n-1] \rrbracket$.
    - Parties input $\llbracket \mathcal{H}(x[0:n-1]) \rrbracket = \llbracket \mathcal{H}(x) \rrbracket[0{:}2^{n-1}] \oplus \llbracket \mathcal{H}(x) \rrbracket[2^{n-1}{:}2^n]$.
    - E inputs $x[0:n-1]$.
    - Parties receive $\llbracket r'(x[0:n-1]) \rrbracket$, and G learns r'.
  - G computes and outputs:
  $$\mathcal{T}(r) \triangleq \mathcal{T}(\hat{r}) \oplus (\mathcal{T}(r') || \mathcal{T}(r'))$$
    The concatenation $\mathcal{T}(r') || \mathcal{T}(r')$ is along the rows.
  - Parties compute and output:
  $$\llbracket r(x) \rrbracket \triangleq \llbracket \hat{r}(x) \rrbracket \oplus \llbracket r'(x[0:n-1]) \rrbracket$$

Figure 4: The core of our approach allows G and E to efficiently evaluate a uniformly random function r such that E does not know r. We compose r from n "half-hidden" uniform functions $\hat{r}$ (see Figure 5). For each function $\hat{r}$, E learns half of the corresponding truth table, but these tables are XORed together in such a way that E learns nothing about r.

- *For each output wire $y[i]$, let $Y[i]$ be the output zero label. Assemble the output decoding string d as a vector of pairs, where for each index i the pair is specified as:*

$$d[i] = (H(\nu, Y[i]) \mid\mid \mathsf{lsb}(Y[i]), H(\nu, Y[i] \oplus \Delta) \mid\mid \mathsf{lsb}(Y[i] \oplus \Delta))$$

Here, ν is a fresh nonce. This choice of d ensures that E will be able to compute only one entry of each pair. (We include the least significant bit of each label to ensure perfect correctness.)

- Input:
  - Parties input a shared one-hot vector $\llbracket \mathcal{H}(x) \rrbracket$ and a sharing $\llbracket x \rrbracket$, where $x \in \{0,1\}^n$.
  - E inputs x.
- Output:
  - G outputs function $\hat{r} : \{0,1\}^n \to \{0,1\}^m$ with uniform truth table $\mathcal{T}(\hat{r})$.
  - E outputs $\mathcal{T}(\hat{r})[0 : 2^{n-1}]$ if $x[n-1]$ is 1; else $\mathcal{T}(\hat{r})[2^{n-1} : 2^n]$.
  - Parties output a sharing $\llbracket \hat{r}(x) \rrbracket$.
- Communication: G sends to E $m\kappa$ bits.
- Computation: Each party uses $O(2^n m\kappa + mc_\kappa + 2^n mc_\kappa/\kappa)$ computation.
- Procedure:
  - Let $\langle\langle Y, Y \oplus x[n-1]\Delta \rangle\rangle = \llbracket x[n-1] \rrbracket$.
  - Let $\langle\langle X_G, X_E \rangle\rangle = \llbracket \mathcal{H}(x) \rrbracket$.
  - $\hat{r} : \{0,1\}^n \to \{0,1\}^m$ is a (not-yet-defined) uniform function and $\hat{R} \triangleq \mathcal{T}(\hat{r})$.
  - We define the left and right halves of vectors:

$$\begin{aligned} \hat{R}^\ell &\triangleq \hat{R}[0 : 2^{n-1}] & \hat{R}^r &\triangleq \hat{R}[2^{n-1} : 2^n] \\ X_G^\ell &\triangleq X_G[0 : 2^{n-1}] & X_G^r &\triangleq X_G[2^{n-1} : 2^n] \\ X_E^\ell &\triangleq X_E[0 : 2^{n-1}] & X_E^r &\triangleq X_E[2^{n-1} : 2^n] \\ \mathcal{H}(x)^\ell &\triangleq \mathcal{H}(x)[0 : 2^{n-1}] & \mathcal{H}(x)^r &\triangleq \mathcal{H}(x)[2^{n-1} : 2^n] \end{aligned}$$

  - G defines $\hat{r}$ by hashing labels Y and $Y \oplus \Delta$: $\hat{R}^\ell \triangleq H(v_{\hat{r}}, Y), \hat{R}^r \triangleq H(v_{\hat{r}}, Y \oplus \Delta)$
  - G defines the following two length-$m\kappa$ strings (row reduction):

$$Z \triangleq H(v_{\mathsf{row}}, Y) \oplus ((\hat{R}^r)^\intercal \cdot X_G^r) \qquad \mathsf{row} \triangleq H(v_{\mathsf{row}}, Y \oplus \Delta) \oplus ((\hat{R}^\ell)^\intercal \cdot X_G^\ell) \oplus Z$$

  - G sends row to E and sets his output share $(\mathcal{T}(\hat{r}))^\intercal \cdot X_G \oplus Z$.
  - E computes:

$$\begin{aligned} &\begin{cases} (H(v_{\hat{r}}, Y)^\intercal \cdot X_E^\ell) \oplus H(v_{\mathsf{row}}, Y) & \text{if } x[n-1] = 0 \\ (H(v_{\hat{r}}, Y \oplus \Delta)^\intercal \cdot X_E^r) \oplus H(v_{\mathsf{row}}, Y \oplus \Delta) \oplus \mathsf{row} & \text{otherwise} \end{cases} \\ &= \begin{cases} ((\hat{R}^\ell)^\intercal \cdot X_E^\ell) \oplus (((\hat{R}^r)^\intercal \cdot X_G^r) \oplus Z) & \text{if } x[n-1] = 0 \\ ((\hat{R}^r)^\intercal \cdot X_E^r) \oplus (((\hat{R}^\ell)^\intercal \cdot X_G^\ell) \oplus Z) & \text{otherwise} \end{cases} \\ &= \begin{cases} ((\hat{R}^\ell)^\intercal \cdot X_E^\ell) \oplus ((\hat{R}^r)^\intercal \cdot X_E^r) \oplus Z & \text{if } x[n-1] = 0 \\ ((\hat{R}^r)^\intercal \cdot X_E^r) \oplus ((\hat{R}^\ell)^\intercal \cdot X_E^\ell) \oplus Z & \text{otherwise} \end{cases} \\ &= (\hat{R}^\intercal \cdot X_E) \oplus Z = (\mathcal{T}(\hat{r})^\intercal \cdot X_E) \oplus Z \end{aligned}$$

  - Parties output: $\langle\langle (\mathcal{T}(\hat{r})^\intercal \cdot X_G) \oplus Z, (\mathcal{T}(\hat{r})^\intercal \cdot X_E) \oplus Z \rangle\rangle = \llbracket (\mathcal{T}(\hat{r})^\intercal \cdot \mathcal{H}(x)) \rrbracket = \llbracket \hat{r}(x) \rrbracket$

Figure 5: Our low-level primitive allows "half-hidden uniform function evaluation". The parties output $\llbracket \hat{r}(x) \rrbracket$ for input x; G outputs uniform function $\hat{r} \in \{0,1\}^n \to \{0,1\}^m$ and E outputs "half" of $\hat{r}$.

- *Output* $(\mathcal{M}, e, d)$.

- *En*(e, x) *is a standard projective procedure. For each input bit* $x[i]$*, En outputs a label appropriate for evaluation:*
$$e[i][x[i]] = X[i] \oplus x[i]\Delta$$
- *Ev*$(\mathcal{M}, X)$ *steps through the circuit gate by gate, using* $\mathcal{M}$ *to map gate input labels to gate output labels. Namely, the procedure proceeds as follows: For each XOR gate, XOR the input labels together [25]. For each AND gate, run the AND gate evaluation procedure formalized by [41]. For each LUT gate, run E's procedure described in Figure 3. The procedure collects each output wire label* $Y[i]$ *and outputs* Y.
- *De*(d, Y) *is a standard projective procedure. For each of E's output labels* $Y[i]$*, the procedure computes:*
$$y[i] = \begin{cases} 0 & \textit{if } (H(\nu, Y[i]) \mid\mid \mathsf{lsb}(Y[i])) = d[i][0] \\ 1 & \textit{if } (H(\nu, Y[i]) \mid\mid \mathsf{lsb}(Y[i])) = d[i][1] \\ \bot & \textit{otherwise} \end{cases}$$
If any *label* $Y[i]$ *fails to decode (i.e., above computes* $\bot$*), then the procedure simply outputs* $\bot$*. Otherwise, the procedure outputs the decoded string* y*. Inclusion of lsbs in the output decoding string ensures* perfect correctness*. (Namely,* $\bot$ *will never arise, unless malicious E tries to forge an output.)*

Compatibility with other garbling techniques. Construction 1 provides the essential interfaces and functionalities for traditional two-input one-output Boolean gates as well as arbitrary n-input m-output lookup tables.

logrow is compatible with many modern techniques in GC, including state-of-the-art Boolean gates [37], GRAM, one-hot accelerated operations, and SGC. Amongst these, compatibility with SGC is by far the most complex, since it requires proving an additional security property. *Strong Stackability* enforces that the garbling of a circuit "look uniform", which allows SGC to safely stack these garblings together [14]. Theorem 6 proves our scheme is strongly stackable.

Hiding LUTs from E**.** Formally, [3] require the definition of a *side-information function* Φ. Given a particular circuit $\mathcal{C}$, this function specifies what information about $\mathcal{C}$ is made available to E. In typical GC constructions, this side-information function is *trivial* in the sense that E is allowed to see the entire circuit, so the side-information function is often omitted from formal discussion.

In our construction, however, we can be stricter and hide from E the specification of each LUT gate function. Thus we give an explicit definition for Φ:

Definition 6. *For a circuit* $\mathcal{C}$ *with XOR gates, AND gates, and LUT gates, we define the* side-information function $\Phi(\mathcal{C})$ *to be the circuit topology and the type of each gate.* $\Phi(\mathcal{C})$ *explicitly* does not *include the function* f *of each LUT gate.*

6 Performance

We argue Construction 1 achieves our claimed performance. Namely, each LUT gate transmits roughly $nm\kappa + Nm$ bits and requires $O(2^n c_\kappa + 2^n m\kappa + nmc_\kappa + 2^n mc_\kappa/\kappa)$ computation from each party. Figure 6 plots our communication consumption as compared to a basic H-row encrypted truth table.

Theorem 1. *In* logrow*, consider a LUT gate with function* $f : \{0,1\}^n \to \{0,1\}^m$*. Each such gate incurs the following cost:*

- $2^n m + (n-1)\kappa + nm\kappa$ *bits of communication.*
- $O(2^n c_\kappa + 2^n m\kappa + nmc_\kappa + 2^n mc_\kappa/\kappa)$ *computation.*

Proof. By inspection of Figures 1 and 3 to 5.

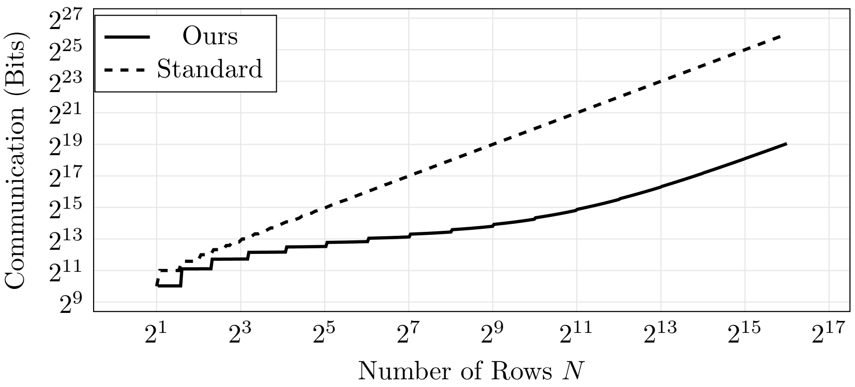

Figure 6: logrow's communication cost for a single LUT gate. We assume $m = 8$ column in the LUT and $\kappa = 128$. Our approach consistently outperforms the (standard) Yao's encrypted truth table approach by orders of magnitude. For $N > 97$, our communication is more than 10× lower than that of the standard approach. For $N > 2^{13}$, our communication is more than 100× lower.

Communication Cost. Recall that we compute $[\![a]\!] \mapsto [\![f(a)]\!]$. There are three steps in our construction where G must transmit information to E:

- G sends the truth table for $f'(x) \triangleq f(x \oplus \alpha) \oplus r(x)$. **Cost:** Nm bits.
- G and E compute $[\![\mathcal{H}(x)]\!]$. **Cost:** $(n-1)\kappa$ bits [13].
- G sends n strings each of length $m\kappa$ to implement n half-hidden uniform functions (Section 4.2). **Cost:** $nm\kappa$ bits.

In total, G must send $Nm + (n-1)\kappa + nm\kappa$ bits. Figure 6 demonstrates that our protocol significantly outperforms a standard encrypted truth table.

Computation Cost. Below, we count the computation cost of each component of our construction.

- The parties first compute $[\![\mathcal{H}(x)]\!]$. **Cost:** $O(2^n c_\kappa)$ computation [13].
- The parties compute a random function $[\![r(x)]\!] = \bigoplus_{i \in [n+1]} [\![r_i(x)]\!]$. Each half hidden random function $[\![r_i(x)]\!]$ incurs the following computation cost:
  - Derive the (2^{n-i-1})-bit strings R_i^ℓ / R_i^r by hashing. **Cost:** $O(2^{n-i} m c_\kappa / \kappa)$ computation.
  - Compute $[\![\mathcal{H}(x[0:n-i])]\!]$ via linear map. **Cost:** $O(2^{n-i}\kappa)$ computation.
  - Construct (resp. decrypt) a garbled row:

$$Z \triangleq H(v_{\mathsf{row}}, Y) \oplus ((\hat{R}^r)^\intercal \cdot X_G^r)$$
$$\mathsf{row} \triangleq H(v_{\mathsf{row}}, Y \oplus \Delta) \oplus ((\hat{R}^\ell)^\intercal \cdot X_G^\ell) \oplus Z$$

    Cost: $O(mc_\kappa)$.
  - Compute the following sum:

$$[\![r_i(x)]\!] = ((R_i^\ell)^\intercal \cdot [\![\mathcal{H}(x[0:n-i])^\ell]\!]) \oplus ((R_i^r)^\intercal \cdot [\![\mathcal{H}(x[0:n-i])^r]\!]) \oplus Z$$

    Cost: $O(2^{n-i} m\kappa)$ computation.

  It takes $O(2^{n-i} m c_\kappa/\kappa + mc_\kappa + 2^{n-i} m\kappa)$ computation to compute $[\![r_i(x)]\!]$. Altogether, the n half hidden random functions **cost**: $O(2^n m c_\kappa/\kappa + nmc_\kappa + 2^n m\kappa)$.

- The parties compute $\llbracket f'(x) \rrbracket = \mathcal{T}(f')^{\intercal} \cdot \llbracket \mathcal{H}(x) \rrbracket$. Both parties use $O(2^n m\kappa)$ bit XORs to compute the matrix product. In addition, G must compute $\mathcal{T}(f')$ from $\mathcal{T}(f)$ and $\mathcal{T}(r_i)$. This uses only $O(2^n m)$ computation. **Cost:** $O(2^n m\kappa)$ computation.
- Finally, the parties compute $\llbracket f(x) \rrbracket = \llbracket f'(x) \rrbracket \oplus \llbracket r(x) \rrbracket$. **Cost:** $O(m\kappa)$.

In total, both G and E use $O(2^n c_\kappa + 2^n m\kappa + nmc_\kappa + 2^n mc_\kappa/\kappa)$ computation. □

7 Security Theorems and Proofs

In this section, we formally state and prove our security claim. Following [3]'s framework for GC, we prove the *correctness*, *obliviousness*, *privacy*, and *authenticity* of logrow. These properties ensure that logrow can instantiate GC-based protocols. logrow is also compatible with Stacked Garbling, which is shown by proving logrow satisfies *strong stackability*.

We note that amongst our proofs, *correctness* and *obliviousness* are the most crucial. For many typical schemes, including ours, *privacy* and *authenticity* follow from obliviousness without much extra effort. Indeed, these proofs are mostly boilerplate, and are similar to proofs given in prior work, e.g. [15]. *Strong stackability* is also relatively straightforward. The most important requirement of strong stackability is that the garbling of a circuit (together with active input labels) be simulatable by an appropriately-sized uniform string. Our obliviousness proof directly shows that this is the case, so our proof of strong stackability is mostly an appeal to our proof of obliviousness. We prove logrow's privacy, obliviousness, and strong stackability to Appendix A, B and C, respectively.

Definition 7 (Correctness)**.** *A garbling scheme is* correct *if for any circuit $\mathcal{C}$ and all inputs x:*

$$De(d, Ev(\mathcal{M}, En(e, x))) = \mathcal{C}(x) \qquad \qquad \textit{where } (\mathcal{M}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$$

Theorem 2. logrow *is correct.*

Proof. By the correctness of individual gates.

Technically, the correctness of each gate proceeds by case analysis. XOR and AND gates are handled using known techniques [25, 41], and so they are correct. Thus, we need only show that our new LUT gates are correct. For the most part, correctness of each LUT gate is argued inline in Figures 3 to 5. In the following, we focus the non-trivial aspects of correctness.

Figure 3 shows that LUT gates are correct, given that Figure 4 indeed (1) delivers random function r to G and (2) delivers shares $\llbracket r(x) \rrbracket$ to G and E.

Correctness of Uniform Function Evaluation (Figure 4). Now, we argue that the recursive uniform function evaluation is correct; namely, the output share $\llbracket r(x) \rrbracket$ is indeed a share of $r(x)$, and G's r agrees on $\llbracket r(x) \rrbracket$. Because Figure 4 proceeds recursively, our proof proceeds by induction on n.

In the base case $n = 0$, G learns $r(\bot) = s$, and the parties output $\langle\langle s\Delta, 0 \rangle\rangle$, which is a correct sharing by construction.

In the inductive case $n > 0$, we can assume that the recursive evaluation is correct: the parties indeed hold shares $\llbracket r'(x) \rrbracket$ that agree with G's function $r' : \{0,1\}^{n-1} \to \{0,1\}^m$. Additionally, Figure 5 indeed correctly computes a half-hidden uniform function $\hat{r}$; correctness is argued inline.

To complete the inductive step, we closely examine how G defines r. Recall that he defines the truth table $\mathcal{T}(r)$ as follows:

$$\mathcal{T}(r) \triangleq \mathcal{T}(r') \oplus (\mathcal{T}(\hat{r}) || \mathcal{T}(\hat{r}))$$

Converting this to a function definition, we know that for any $a \in \{0,1\}^n$

$$\begin{aligned}
r(a) &= \mathcal{T}(r)^{\mathsf{T}} \cdot \mathcal{H}(a) \\
&= (\mathcal{T}(\hat{r}) \oplus (\mathcal{T}(r')||\mathcal{T}(r'))^{\mathsf{T}} \cdot \mathcal{H}(a)) \\
&= (\mathcal{T}(\hat{r})^{\mathsf{T}} \cdot \mathcal{H}(a)) \oplus (\mathcal{T}(r')^{\mathsf{T}} \cdot \mathcal{H}(a)[0 : 2^{n-1}]) \oplus (\mathcal{T}(r')^{\mathsf{T}} \cdot \mathcal{H}(a)[2^{n-1} : 2^n]) \\
&= (\mathcal{T}(\hat{r})^{\mathsf{T}} \cdot \mathcal{H}(a)) \oplus (\mathcal{T}(r')^{\mathsf{T}} \cdot (\mathcal{H}(a)[0 : 2^{n-1}] \oplus \mathcal{H}(a)[2^{n-1} : 2^n])) \\
&= (\mathcal{T}(\hat{r})^{\mathsf{T}} \cdot \mathcal{H}(a)) \oplus (\mathcal{T}(r')^{\mathsf{T}} \cdot \mathcal{H}(a[0 : n-1])) \\
&= \hat{r}(a) \oplus r'(a[0 : n-1])
\end{aligned}$$

This matches the parties computed shares $[\![r(x)]\!] = [\![\hat{r}(x)]\!] \oplus [\![r'(x[0 : n-1])]\!]$, so we can conclude that G's function r agrees with $[\![r(x)]\!]$. By induction, the recursive uniform function evaluation is correct, and so LUT gates are correct.

Since each gate type is correct, logrow is correct. □

Definition 8 (Obliviousness)**.** *A garbling scheme is* oblivious *if for any circuit $\mathcal{C}$ and for all inputs x there exists a simulator $\mathcal{S}_{\mathsf{obv}}$ such that the following computational indistinguishability holds:*

$$(\mathcal{M}, X) \overset{c}{\approx} \mathcal{S}_{\mathsf{obv}}(1^\kappa, \Phi(\mathcal{C})) \qquad \textit{where } (\mathcal{M}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C}) \textit{ and } X \leftarrow En(e, x)$$

Theorem 3. logrow *is oblivious.*

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

Obliviousness is arguably the most important GC security notion, as it ensures that the GC alone conveys no information to the evaluator.

At a high level, $\mathcal{S}_{\mathsf{obv}}$ proceeds gate-by-gate through the circuit, at each gate simulating appropriate GC material. To handle AND/XOR gates, our simulator simply calls out to gate simulators defined by prior work [41].

The non-standard part of the simulation – and our focus in this proof – is the handling of a single LUT gate. At a high level, this simulation is straightforward. E receives only three kinds of messages from G:

- A truth table $\mathcal{T}(f')$ (see Figure 3). $\mathcal{T}(f')$ is masked by a random function r, and so can by simulated by a uniform matrix of appropriate dimension.
- $n-1$ garbled rows needed to construct the one-hot encoding $[\![\mathcal{H}(x)]\!]$. [13] demonstrated that each of these length-κ rows can be simulated by uniform strings of length κ.
- n garbled rows each of length $m\kappa$. Each of these rows is encrypted by H, and the definition of circular correlation robustness (Definition 1) is sufficient to ensure that these can also be simulated by uniform bits.

In short, each LUT gate is simulated by a uniform string of appropriate length.

In more detail, $\mathcal{S}_{\mathsf{obv}}$ proceeds as follows. First, we simulate the label for each input bit $x[i]$ with a uniform string $\hat{X}[i]$. This is indistinguishable because the real label $X[i]$ (or $X[i] \oplus \Delta$) is also sampled uniformly.

$\mathcal{S}_{\mathsf{obv}}$ then proceeds gate-by-gate through $\mathcal{C}$. Note, the topology of $\mathcal{C}$ is explicitly contained in $\Phi(\mathcal{C})$ (Definition 6). At each gate, we use the simulated gate input labels to simulate appropriate material and gate output labels. After simulating each gate, $\mathcal{S}_{\mathsf{obv}}$ returns $(\hat{\mathcal{M}}, \hat{Y})$ where $\hat{\mathcal{M}}$ is the collection of gate material and $\hat{Y}$ is the collection of simulated labels on circuit output wires. The core task of $\mathcal{S}_{\mathsf{obv}}$ is to proceed by case analysis on each gate.

XOR gates are 'free,' since no material is required. To simulate the gate, $\mathcal{S}_{\mathsf{obv}}$ simply XORs the simulated input labels, places the result on the output wire, and then continues to the next gate.

AND gates are implemented by the half-gate technique [41]. [41, p.13] provides an explicit simulator procedure for AND gates which we call which we call $\mathcal{S}_{\mathsf{AND}}$. For each AND gate, $\mathcal{S}_{\mathsf{obv}}$ feeds the input labels to $\mathcal{S}_{\mathsf{AND}}$, obtains the output label, and appends the two ciphertexts (for the half gates) to $\hat{\mathcal{M}}$.

LUT gates require G to deliver three kinds of material to E (see high level summary above). We argue that $\mathcal{S}_{\mathsf{obv}}$ can simulate all of this material with uniform bits.

- G sends to E the material for $[\![\mathcal{H}(x)]\!]$. [13, p.15] provides an explicit simulator for simulating the one-hot garbling procedure listed in Figure 1. We call this simulator $\mathcal{S}_{\mathcal{H}}$. $\mathcal{S}_{\mathsf{obv}}$ invokes $\mathcal{S}_{\mathcal{H}}$ on simulated input labels, resulting in simulated material and simulated output labels. It appends the resultant material to $\hat{\mathcal{M}}$.

- G sends one length $m\kappa$ string to E for each of n half-hidden uniform function evaluations (Figure 5). To simulate each of these strings, $\mathcal{S}_{\mathsf{obv}}$ samples a $m\kappa$-bit uniform string $\hat{Z}$ and appends it to $\hat{\mathcal{M}}$. We argue that each of these m bits is indistinguishable. Recall that G sends the following to E:

  $$H(v_{\mathsf{row}}, Y \oplus \Delta) \oplus ((\hat{R}^{\ell})^{\intercal} \cdot X_G^{\ell}) \oplus H(v_{\mathsf{row}}, Y) \oplus ((\hat{R}^{r})^{\intercal} \cdot X_G^{r})$$

  (Here, we have inlined the definition of Z listed in Figure 5.) Let $\hat{Y}$ be the simulated input label for Y and let $\hat{X}$ be the simulated garbling of the one-hot vector $[\![\mathcal{H}(x)]\!]$. WLOG, suppose the simulated value $x[n-1]$ is 0; the indistinguishability argument for $x[n-1] = 1$ is symmetric. We also define the following:

  $$L \triangleq ((\hat{R}^{\ell})^{\intercal} \cdot X_G^{\ell}) \qquad R \triangleq ((\hat{R}^{r})^{\intercal} \cdot X_G^{r})$$

  Note the following indistinguishability (which holds even in the context of appropriate input labels):

  $$\begin{aligned} \hat{Y} &\overset{c}{\approx} \hat{Y} \oplus L \oplus H(v_{\mathsf{row}}, Y) \oplus R && \hat{Y} \text{ is a one-time pad} \\ &\overset{c}{\approx} \mathcal{R}(\nu_{\mathsf{row}}, Y, 0) \oplus L \oplus H(v_{\mathsf{row}}, Y) \oplus R && \mathcal{R} \text{ is a random function} \\ &\overset{c}{\approx} \mathit{circ}_{\Delta}(\nu_{\mathsf{row}}, Y, 0) \oplus L \oplus H(v_{\mathsf{row}}, Y) \oplus R && \text{Definition 1} \\ &\overset{c}{\approx} H(v_{\mathsf{row}}, Y \oplus \Delta) \oplus L \oplus H(v_{\mathsf{row}}, Y) \oplus R && \text{Real} \end{aligned}$$

  In short, the inclusion of the call to H on one label that E does not know ensures that the row appears uniform.

- G sends to E the truth table for $f'(x) \triangleq f(x \oplus \alpha) \oplus r(x)$. $\mathcal{S}_{\mathsf{obv}}$ simulates this truth table by appending to $\hat{\mathcal{M}}$ an appropriately sized ($2^n m$-bits) uniform string. Since $\mathcal{T}(f') = \mathcal{T}(f) \oplus \mathcal{T}(r)$, as long as we can show the uniformity of $\mathcal{T}(r)$, $\mathcal{T}(f')$ is also uniform.

  One tricky issue is that $r(x) \triangleq \bigoplus_{i=0}^{n} r_i(x[0 : n-i])$, and the adversary knows half of each $r_{i<n}$. Still, we can show that for any $a \in \{0,1\}^n$, $r(a)$ is uniformly distributed regardless of the adversary's knowledge of $r_{i<n}$.

  We prove this by showing that for any a, $r(a)$ is masked by a uniform value $r_i(a)$ that is *not* known to the adversary. Importantly, this value is not "reused" to mask any other $r(\cdot)$, so the distribution of the full string r is uniform. We show this by induction.

  In the base case $n = 0$, $r(\bot) = s$ is a uniform value only known to G, and hence unavailable to the adversary. I.e., $\mathcal{S}_{\mathsf{obv}}$ can simulate s by a uniform string. **In the inductive case** $n' < n$, $\mathcal{S}_{\mathsf{obv}}$ can simulate r' by a uniform string. Now we show that a uniform string can also simulate $r(x) \triangleq \hat{r}(x) \oplus r'(x)$. For $a \in \{0,1\}^n$, there are two possible cases:

  - $a[n-1] = x[n-1]$. By the uniformity of r', $r'(a[0 : n-1])$ is uniform. Thus, $r(a) = \hat{r}(a) \oplus r'(a[0 : n-1])$ is uniform.

- $a[n-1] = x[n-1] \oplus 1$. Since the adversary does not have the label for $[\![x[n-1] \oplus 1]\!]$, and $r(a)$ is generated by hashing the label of $[\![x[n-1] \oplus 1]\!]$ that the adversary does not have, $r(a)$ is uniform.

Notice that for each a, $r(a)$ can be independently simulated by a uniform string. Hence, $\mathcal{T}(r)$, defined as the concatenation of these strings, is uniform.

Thus, the full LUT gate can be simulated simply by drawing uniform strings of appropriate length.

From here, the proof of indistinguishability follows a basic hybrid argument, similar to the standard proof of GC security given by [27].

logrow is oblivious. □

Definition 9 (Privacy)**.** *A garbling scheme is* private *if for any circuit $\mathcal{C}$ and for all inputs x there exists a simulator $\mathcal{S}_{\mathsf{prv}}$ such that the following computational indistinguishability holds*

$$(\mathcal{M}, X, d) \overset{c}{\approx} \mathcal{S}_{\mathsf{prv}}(1^\kappa, y, \Phi(\mathcal{C}))$$

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

Theorem 4. logrow *is private.*

We refer readers to Appendix A for the proof.

Definition 10 (Authenticity)**.** *A garbling scheme is* authentic *if for all circuits $\mathcal{C}$, all inputs x, and all probabilistic polynomial time (PPT) adversaries $\mathcal{A}$, the following probability is negligible in κ:*

$$\mathsf{Pr}[Ev(\mathcal{M}, x) \neq y' \wedge De(d, y') \neq \bot]$$

where $(\mathcal{M}, e, d) \leftarrow Gb(1^\kappa, c)$, $x \leftarrow En(e, x)$, and $y' \leftarrow \mathcal{A}(\mathcal{C}, \mathcal{M}, x)$.

Authenticity ensures that even a malicious evaluator $\mathcal{A}$ cannot compute output labels that successfully decode, except by running the GC as intended.

Notably, $\mathcal{A}$ is given the full circuit description $\mathcal{C}$, not just the side-information function $\phi(\mathcal{C})$. Thus, $\mathcal{A}$ has access to the function of each LUT gate. This captures scenarios where the evaluator may have side information about each LUT.

Theorem 5. logrow *is authentic.*

We refer readers to Appendix B for the proof.

Acknowledgements. This research was developed with funding from a Visa research award, a Cisco research award, a USDA APHIS research award (under opportunity number USDA-APHIS-10025-VSSP0000-23-0003), and NSF awards CNS-2246353, CNS-2246354, and CCF-2217070.

References

[1] Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (Aug 1992). https://doi.org/10.1007/3-540-46766-1_34

[2] Beimel, A., Kushilevitz, E., Nissim, P.: The complexity of multiparty PSM protocols and related models. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 287–318. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_10

[3] Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012. pp. 784–796. ACM Press (Oct 2012). https://doi.org/10.1145/2382196.2382279

[4] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_16

[5] Bruggemann, A., Hundt, R., Schneider, T., Suresh, A., Yalame, H.: Flute: Fast and secure lookup table evaluations. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 515–533. IEEE Computer Society, Los Alamitos, CA, USA (may 2023). https://doi.org/10.1109/SP46215.2023.10179345, https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179345

[6] Choi, S.G., Katz, J., Kumaresan, R., Zhou, H.S.: On the security of the "free-XOR" technique. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 39–53. Springer, Heidelberg (Mar 2012). https://doi.org/10.1007/978-3-642-28914-9_3

[7] Dessouky, G., Koushanfar, F., Sadeghi, A.R., Schneider, T., Zeitouni, S., Zohner, M.: Pushing the communication barrier in secure computation using lookup tables. In: NDSS 2017. The Internet Society (Feb / Mar 2017)

[8] Feige, U., Kilian, J., Naor, M.: A minimal model for secure computation (extended abstract). In: 26th ACM STOC. pp. 554–563. ACM Press (May 1994). https://doi.org/10.1145/195058.195408

[9] Haque, A., Heath, D., Kolesnikov, V., Lu, S., Ostrovsky, R., Shah, A.: Garbled circuits with sublinear evaluator. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part I. LNCS, vol. 13275, pp. 37–64. Springer, Heidelberg (May / Jun 2022). https://doi.org/10.1007/978-3-031-06944-4_2

[10] Heath, D.: New Directions in Garbled Circuits. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, USA (2022), http://hdl.handle.net/1853/66604

[11] Heath, D.: Efficient arithmetic in garbled circuits. Cryptology ePrint Archive, Paper 2024/139 (2024), https://eprint.iacr.org/2024/139

[12] Heath, D., Kolesnikov, V.: Stacked garbling - garbled circuit proportional to longest execution path. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 763–792. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56880-1_27

[13] Heath, D., Kolesnikov, V.: One hot garbling. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 574–593. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484764

[14] Heath, D., Kolesnikov, V.: LogStack: Stacked garbling with $O(b \log b)$ computation. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part III. LNCS, vol. 12698, pp. 3–32. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-77883-5_1

[15] Heath, D., Kolesnikov, V., Ostrovsky, R.: EpiGRAM: Practical garbled RAM. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part I. LNCS, vol. 13275, pp. 3–33. Springer, Heidelberg (May / Jun 2022). https://doi.org/10.1007/978-3-031-06944-4_1

[16] Hong, C., Katz, J., Kolesnikov, V., Lu, W., Wang, X.: Covert security with public verifiability: Faster, leaner, and simpler. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part III. LNCS, vol. 11478, pp. 97–121. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17659-4_4

[17] Huang, Y., Katz, J., Kolesnikov, V., Kumaresan, R., Malozemoff, A.J.: Amortizing garbled circuits. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 458–475. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44381-1_26

[18] Ishai, Y., Kushilevitz, E.: Private simultaneous messages protocols with applications. In: Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems. pp. 174–183 (1997). https://doi.org/10.1109/ISTCS.1997.595170

[19] Ishai, Y., Kushilevitz, E.: Randomizing polynomials: A new representation with applications to round-efficient secure computation. In: 41st FOCS. pp. 294–304. IEEE Computer Society Press (Nov 2000). https://doi.org/10.1109/SFCS.2000.892118

[20] Ishai, Y., Kushilevitz, E., Meldgaard, S., Orlandi, C., Paskin-Cherniavsky, A.: On the power of correlated randomness in secure computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 600–620. Springer, Heidelberg (Mar 2013). https://doi.org/10.1007/978-3-642-36594-2_34

[21] Kennedy, W.S., Kolesnikov, V., Wilfong, G.T.: Overlaying conditional circuit clauses for secure computation. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 499–528. Springer, Heidelberg (Dec 2017). https://doi.org/10.1007/978-3-319-70697-9_18

[22] Kolesnikov, V.: Free IF: How to omit inactive branches and implement S-universal garbled circuit (almost) for free. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part III. LNCS, vol. 11274, pp. 34–58. Springer, Heidelberg (Dec 2018). https://doi.org/10.1007/978-3-030-03332-3_2

[23] Kolesnikov, V., Kumaresan, R.: Improved OT extension for transferring short secrets. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 54–70. Springer, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40084-1_4

[24] Kolesnikov, V., Mohassel, P., Rosulek, M.: FleXOR: Flexible garbling for XOR gates that beats free-XOR. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 440–457. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44381-1_25

[25] Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (Jul 2008). https://doi.org/10.1007/978-3-540-70583-3_40

[26] Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (May 2007). https://doi.org/10.1007/978-3-540-72540-4_4

[27] Lindell, Y., Pinkas, B.: A proof of security of Yao's protocol for two-party computation. Journal of Cryptology **22**(2), 161–188 (Apr 2009). https://doi.org/10.1007/s00145-008-9036-8

[28] Lindell, Y., Riva, B.: Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 476–494. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44381-1_27

[29] Lu, S., Ostrovsky, R.: How to garble RAM programs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (May 2013). https://doi.org/10.1007/978-3-642-38348-9_42

[30] Naor, M., Nissim, K.: Communication preserving protocols for secure function evaluation. In: 33rd ACM STOC. pp. 590–599. ACM Press (Jul 2001). https://doi.org/10.1145/380752.380855

[31] Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM conference on Electronic commerce. pp. 129–139. ACM (1999)

[32] Park, A., Lin, W.K., Shi, E.: NanoGRAM: Garbled RAM with $\widetilde{O}(\log N)$ overhead. Cryptology ePrint Archive, Report 2022/191 (2022), `https://eprint.iacr.org/2022/191`

[33] Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (Dec 2009). https://doi.org/10.1007/978-3-642-10366-7_15

[34] Rathee, D., Bhattacharya, A., Gupta, D., Sharma, R., Song, D.: Secure floating-point training. Cryptology ePrint Archive (2023)

[35] Rathee, D., Bhattacharya, A., Sharma, R., Gupta, D., Chandran, N., Rastogi, A.: SecFloat: Accurate floating-point meets secure 2-party computation. In: 2022 IEEE Symposium on Security and Privacy. pp. 576–595. IEEE Computer Society Press (May 2022). https://doi.org/10.1109/SP46214.2022.9833697

[36] Rathee, D., Rathee, M., Goli, R.K.K., Gupta, D., Sharma, R., Chandran, N., Rastogi, A.: SiRnn: A math library for secure RNN inference. In: 2021 IEEE Symposium on Security and Privacy. pp. 1003–1020. IEEE Computer Society Press (May 2021). https://doi.org/10.1109/SP40001.2021.00086

[37] Rosulek, M., Roy, L.: Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part I. LNCS, vol. 12825, pp. 94–124. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84242-0_5

[38] Troncoso-Pastoriza, J.R., Katzenbeisser, S., Celik, M.: Privacy preserving error resilient dna searching through oblivious automata. In: Ning, P., De Capitani di Vimercati, S., Syverson, P.F. (eds.) ACM CCS 2007. pp. 519–528. ACM Press (Oct 2007). https://doi.org/10.1145/1315245.1315309

[39] Yang, Y., Peceny, S., Heath, D., Kolesnikov, V.: Towards generic MPC compilers via variable instruction set architectures (visas). IACR Cryptol. ePrint Arch. p. 953 (2023), https://eprint.iacr.org/2023/953

[40] Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press (Oct 1986). https://doi.org/10.1109/SFCS.1986.25

[41] Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_8

Appendices

A Proof of Privacy (Theorem 4)

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

At a high level, privacy follows from obliviousness and from the definition of De. The key challenge of privacy is in demonstrating that our simulation can account for the output value y, which in the real world is made available to the adversary by the output decoding string d. To handle this, $\mathcal{S}_{\mathsf{prv}}$ uses $\mathcal{S}_{\mathsf{obv}}$ to simulate a garbled circuit and then simulates a garbled output decoding string $\hat{d}$ that causes the simulated garbled circuit to output the correct value.

In more detail, we construct $\mathcal{S}_{\mathsf{prv}}$ as follows. $\mathcal{S}_{\mathsf{prv}}$ first invokes $\mathcal{S}_{\mathsf{obv}}(1^\kappa, \Phi(\mathcal{C}))$ to obtain simulated material $\hat{\mathcal{M}}$ and simulated input labels $\hat{X}$. $\mathcal{S}_{\mathsf{prv}}$ then *evaluates* this fake garbled circuit by invoking $Ev(\hat{\mathcal{M}}, \hat{X})$, yielding output labels $\hat{Y}$. Next, $\mathcal{S}_{\mathsf{prv}}$ tailors an output decoding string that precisely causes $\hat{Y}$ to decrypt to y. I.e., consider each output label $\hat{Y}[i]$ with corresponding output $y[i]$. WLOG, assume $y[i] = 0$. The 1 case is symmetric. $\mathcal{S}_{\mathsf{prv}}$ constructs $\hat{d}$ as follows:

$$\hat{d}[i] = (H(\nu, \hat{Y}), R) \qquad\qquad \text{where } R \in_{\$} \{0,1\}^\kappa$$

By inspecting the definition of De (Construction 1), we can see that this ensures that the simulated output decodes to the correct value:

$$De(\hat{d}, \hat{Y}) = y$$

At the same time, for each output $\hat{Y}[i]$, the adversary cannot distinguish $d[i][1]$ from $\hat{d}[i][1]$ (again, WLOG assuming $y[i] = 0$). This holds by the properties of the hash function H and by the fact that in the real world obliviousness prevents the adversary from obtaining $Y[i] \oplus \Delta$. I.e., $H(\nu, Y[i] \oplus \Delta)$ appears uniform.

$\mathcal{S}_{\mathsf{prv}}$ outputs $(\hat{\mathcal{M}}, \hat{X}, \hat{d})$.

logrow is private. □

B Proof of Authenticity (Theorem 5)

Proof. By the definition of $\mathcal{S}_{\mathsf{prv}}$ and of De.

At a high level, authenticity follows almost immediately from the definition of $\mathcal{S}_{\mathsf{prv}}$. Privacy ensures that *no* PPT algorithm can distinguish a real garbled circuit from the privacy simulation. But $\mathcal{S}_{\mathsf{prv}}$'s output decoding string $\hat{d}$ consists of labels that are chosen *uniformly at random.* Thus, it *cannot possibly be the case* that $\mathcal{A}$ can attack the authenticity of the privacy simulator: doing so would require $\mathcal{A}$ to guess a uniformly chosen entry of $\hat{d}$. Thus, if some adversary $\mathcal{A}$ *can* attack the real-world GC, then $\mathcal{A}$ can be used to distinguish the real-world GC from the privacy simulation – a contradiction.

In more detail, suppose, for the sake of deriving a contradiction, that $\mathcal{A}$ *can* construct $Y' \neq Y$ that successfully decodes. We use $\mathcal{A}$ to construct a privacy distinguisher:

- Distinguisher $\mathcal{D}^{\mathcal{C}}_{\mathsf{prv}}(\mathcal{M}, X, d)$:
  - Compute $Y = Ev(\mathcal{M}, X)$.
  - Compute $Y' = \mathcal{A}(\mathcal{C}, M, X)$.
  - Output $(Y' \neq Y) \wedge (De(d, Y') \neq \perp)$.

Consider the following:

- Compute $(\hat{\mathcal{M}}, \hat{X}, \hat{d}) \leftarrow \mathcal{S}_{\mathsf{prv}}(1^\kappa, y, \Phi(\mathcal{C})$.
- Compute and output $\mathcal{D}^{\mathcal{C}}_{\mathsf{prv}}(\hat{\mathcal{M}}, \hat{X}, \hat{d})$

The above output must be 0 with overwhelming probability: Recall that for each output bit $y[i]$, $\mathcal{S}_{\mathsf{prv}}$ hashes the simulated output label $H(v, \hat{Y}[i])$ and uses this as an entry of $\hat{d}$. However, $\mathcal{S}_{\mathsf{prv}}$ does not know the other label, so it appends a uniform string R to $\hat{d}$. It is infeasible for $\mathcal{A}$ find a label $Y'[i]$ such that $H(\nu, Y'[i]) = R$.

However, if $(\mathcal{M}, X, d)$ is from the real execution, then by our assumption, $\mathcal{A}$ must output a valid label $Y' \neq Y$ such that Y' successfully decodes. Therefore, in the real world $\mathcal{D}^{\mathcal{C}}_{\mathsf{prv}}$ outputs 1 with non-negligible probability. Hence, $\mathcal{D}^{\mathcal{C}}_{\mathsf{prv}}$ is indeed a privacy distinguisher.

We have reached a contradiction. logrow is private, and so $\mathcal{D}^{\mathcal{C}}_{\mathsf{prv}}$ should not exist. Thus, it must be the case that $\mathcal{A}$ does not exist.

logrow is authentic. □

C Proof of Strong Stackability

Definition 11 (Strong Stackability)**.** *A garbling scheme is* strongly stackable *[14] if:*

1. *For all circuits $\mathcal{C}$ and all inputs x,*
$$(\mathcal{C}, \mathcal{M}, En(e, x)) \overset{c}{\approx} (\mathcal{C}, \mathcal{M}', x')$$
where $(\mathcal{M}, e, \cdot) \leftarrow Gb(1^\kappa, \mathcal{C})$, $X' \leftarrow \{0,1\}^{|X|}$, and $\mathcal{M}' \leftarrow \{0,1\}^{|\mathcal{M}|}$.

2. *The scheme is* projective*.*

3. *There exists an efficient deterministic procedure* Color *that maps strings to {0, 1} such that such that for all $\mathcal{C}$ and all projective label pairs $A^0, A^1 \in d$:*
$$\mathsf{Color}(A^0) \neq \mathsf{Color}(A^1)$$
where $(\cdot, \cdot, d) = Gb(1^\kappa, \mathcal{C})$.

4. *There exists an efficient deterministic procedure* Key *that maps strings to $\{0,1\}^\kappa$ such that for all $\mathcal{C}$ and all projective labels pairs $A^0, A^1 \in d$:*
$$\mathsf{Key}(A^0) \mid\mid \mathsf{Key}(A^1) \overset{c}{\approx} \{0,1\}^{2\kappa}$$
where $(\cdot, \cdot, d) = Gb(1^\kappa, \mathcal{C})$.

Theorem 6. logrow *is strongly stackable.*

Proof. We show that logrow satisfies each property.

- Property 1 requires that the GC material and input labels $\mathcal{M}, X$ should be uniform. This follows immediately from our proof of obliviousness (Theorem 3), where we explicitly simulated the GC with uniform bits.

- Properties 2−4 require that the scheme be projective (Property 2) and that each pair of labels have distinct color bits (Property 3). We first note that our scheme is projective by construction.

  Recall our definition of d:
$$d[i] = (H(\nu, Y[i]), H(\nu, Y[i] \oplus \Delta))$$
  Formally, we slightly modify our definition of d to replace the lsb of the first entry of the pair by a zero; similarly, we replace the lsb of the second entry of the pair by a one. Thus, the lsb of each entry is an index into the pair. We define Color to be a function that extracts the lsb. We keep this small change to d here to avoid cluttering our notation elsewhere. Similarly, Key extracts all bits from d other than the LSBs. The pair of key pairs is indistinguishable from uniform by the properties of H.

  In short, strong stackability holds by definition of the obliviousness simulator and by the definition of our encodings.

  logrow is strongly stackable. □