Toss: Garbled PIR from Table-Only Stacking

Lucien K. L. Ng[1] and Vladimir Kolesnikov[2]

[1]kng68@gatech.edu Georgia Institute of Technology
[2]kolesnikov@gatech.edu Georgia Institute of Technology

Abstract

Garbled Circuits (GC) is a foundational primitive for secure two-party computation (2PC). Garbled Private Information Retrieval (GPIR) is a GC technique for looking up a public array or database (DB) on a private index unknown to either player. GPIR immediately implies GC evaluation of functions implemented as a publicly known look-up table (LUT).

GPIR is costly: it can be obtained by a linear scan, adapting Garbled RAM, stacking GC branches implementing access to table elements, and, most recently, from GC Look-up Table logrow (Heath et al., Eurocrypt 2024). For a N-row DB lookup of m-bit rows, logrow's computation is $\approx O(Nm\kappa)$, and communication is $O(m(\log N \cdot \kappa + N))$. logrow thus can be effectively used on tables of sizes up to $\approx 2^{15}$.

We propose Toss, a new efficient GPIR with dramatically reduced bandwidth consumption (a scarce resource in MPC!), both asymptotically and concretely. Our communication cost is $O(\sqrt{N}m\sqrt{\kappa})$ with a small constant, sublinear both in N and security parameter κ. Our computation cost is $O(Nm\kappa + (\sqrt{N/\kappa}m + N)c_\kappa)$, where c_κ is the computational cost of hash evaluation. This computation cost is about the same as or slightly lower than logrow's.

In concrete terms, for a 2^{20}-row LUT of 8-bit items, we improve over logrow by factor $>31\times$ in communication. On a laptop over 100Mbps channel, the throughput increases from $\approx$10.6 lookup/s to $\approx$81 lookup/s, achieving $>7.5\times$ improvement. For 10Mbps channel, Toss's throughput is $>28\times$ better. Communication improvement grows with N: e.g., for $N = 2^{25}, m = 32$, improvement is $> 512\times$.

Toss builds on stacked garbling (SGC) and logrow with multiple nuanced low-level optimizations, requiring reworking of their internals and interfaces. We emphasize that constructing GPIR directly from SGC incurs logarithmic overhead in computation, which actually *decreases* throughput in typical "laptop + LAN" testbeds.

We implement our construction and report on its performance.

Contents

1 Introduction **4**
- 1.1 Contribution and Roadmap 4
- 1.2 Applications 5

2 Related Work **5**

3 Preliminaries **7**
- 3.1 Notation 7
- 3.2 GC Definition 7
  - 3.2.1 Circular Correlation Robustness Hash 7
  - 3.2.2 Garbling Scheme 8
- 3.3 Stacked Garbling 8
- 3.4 LogStack 9
  - 3.4.1 The SortingHat garbled gadget 9
  - 3.4.2 E's unstacking using $\mathcal{SH}$ 9
  - 3.4.3 Logarithmic computational cost 10
- 3.5 One-Hot Garbling 11
- 3.6 GC$-$LUT `logrow` 12
  - 3.6.1 `logrow` 12

4 High-Level Overview **13**
- 4.1 Basic Idea: Stacking GC$-$LUTs 13
- 4.2 Obstacle: LogStack's Overheads 13
- 4.3 Intuition of our Computationally-Efficient Solution, `Toss` 13
  - 4.3.1 E's table-only stacking 14
  - 4.3.2 Improved G's garbage collection 14

5 Lower-level Intuition **15**
- 5.1 SortingHat from Private One-hot Garbling 15
  - 5.1.1 Private One-Hot Garbling 15
  - 5.1.2 SortingHat for `Toss` 17
  - 5.1.3 in-mux for Generic Stacking. 17
- 5.2 Stacking and Unstacking of Sub-Tables 18
- 5.3 Our in-mux for `Toss` 18
  - 5.3.1 Getting Input for the Active Branch 19
  - 5.3.2 Obliviously Distributing Inputs to All Branches 19
- 5.4 E's Efficient Unstacking in `Toss` 19
  - 5.4.1 Computation Cost Accounting for E's Unstacking 19
- 5.5 Garbage Collection 20
  - 5.5.1 Our Custom out-mux 20
  - 5.5.2 Efficient Prediction for Translation 20

6 Formalization and Security Theorem **21**
- 6.1 Formal Constructions of `Toss` 21
- 6.2 Security Theorems and Proofs 22
  - 6.2.1 Security Proofs for Components 26

7 Performance **30**
- 7.1 Proofs of Performance Analysis 30

A Sketch of OuterProduct **35**

B Formal Construction of LogStack **35**

1 Introduction

Garbled Circuit (GC) is a principal technique for two-party computation (2PC). One central feature of GC is that it runs in constant rounds. In the classic GC construction, as well as in modern GC, required bandwidth is usually the limiting resource. Thus, the communication is often the primary performance concern. At the same time, several recent GC techniques, such as stacked GC (SGC) [24, 12, 17], incur significant computational overhead as a trade off for improving communication. As a result, the bottleneck of some of these techniques is in computation. In our work, we show how to significantly improve communication for GPIR, while incurring essentially no penalty in computation compared to prior work.

Several recent GC advances focused on going beyond circuit evaluation. Stacked garbling [24, 14, 13] showed efficient evaluation of conditionals. Garbled RAM (GRAM) [20, 21] proposes an oblivious RAM in the constant-round GC setting which is competitive with prior multi-round ORAM. Garbled Lookup Table (GC−LUT) `logrow` [18] allows a one-time lookup of a table known (only) to the Garbler by a private unknown index. Together, these techniques greatly expand the practical power of GC and bring us closer to efficiently implementing complex tasks such as CPU-emulation-based 2PC [36] and generic GC compilers from high-level languages.

Most related to our work are GRAM and Garbled LUT, efficient techniques for accessing information for garbled computation.

GPIR. Adding to the collection of efficient garbled access primitives, we consider Garbled Private Information Retrieval (GPIR). Here, the parties have an N-row *public* database (large lookup table) and want to load onto GC wire(s) the i-th DB element, based on the garbled index i computed inside GC. In contrast, GC-LUT [18] can handle look-ups in tables hidden from the evaluator.

GPIR is useful for accelerating floating-point computation, privacy-preserving machine learning (PPML), deterministic finite automata (DFA), *e.g.*, for substring search and DNA matching (cf. Section 1.2).

Both GRAM and garbled LUT can be used to achieve GPIR, at a significant cost. Garbled RAM requires a lengthy initialization to put the N entries into the RAM with $O(N)$ writes, requiring $O(N \log^3 N \cdot \kappa)$ bits of communication. This is unjustified if the parties only need to perform a small number of look-ups and thus cannot amortize the initialization cost. Additionally, GRAM operates on larger blocks, usually of size at least logarithmic in memory size (this is needed for efficiency and to implement recursive lookups), while GPIR works well even with single-bit blocks. Further, current Garbled RAM requires sequential computation in performing a look-up, while we can parallelize the computation. Before our work, direct application of garbled LUT `logrow` [18] was the best mechanism to obtain GPIR. Our GPIR is more efficient than the `logrow`-based approach; in particular, we remove `logrow`'s $O(N)$ communication cost and achieve orders of magnitude reduction.

1.1 Contribution and Roadmap

We propose `Toss`, an efficient Garbled PIR gate that can be used within standard GC. Let c_κ be the cost of hash evaluation. Our PIR gate costs $O(\sqrt{N}m\sqrt{\kappa})$ communication with $O(Nm\kappa + (\sqrt{N/\kappa}m + N)c_\kappa)$ computation. Our Construction 1 is constant-round: all gates, including PIR, are garbled and sent to E simultaneously.

At the very high level, we split the DB in $\sqrt{N/\kappa}$ number of chunks and stack their GC evaluation procedures *a-la* Stacked Garbling (SGC). While this improves communication, the overall performance (on high-bandwidth channels) suffers; stacking incurs $O(\log N)$ computation overhead, and computation is already the bottleneck in `logrow` on such channels.

Our crucial low-level technical contribution is a tailored stacking protocol for GPIR, which achieves stacking-like communication savings without the high computational penalty of stacking. That is, our computation is similar to `logrow` both asymptotically and concretely. While stacking LUTs is a natural idea, it is surprisingly tricky to avoid the logarithmic overhead in computation. We could not use the building blocks LogStack, one-hot garbling and `logrow` as-is; it is required to use and rework their underlying ideas, data structures, garbling tricks, and even interfaces.

Approach	Bandwidth	Rounds	Auth	GC-Comp
`Toss` **(Ours)**	$O(\sqrt{N}m\sqrt{\kappa})$	Non-interactive	✓	✓
`logrow`	$O(nm\kappa + Nm)$	Non-interactive	✓	✓
Flute [3]	$O(N + m)$	$O(n)$		
Floram [2, 7]	$O(n\kappa + m)$	$O(n)$		

Table 1: Comparison with 2PC lookup schemes (including interactive) for a table with $N = 2^n$ rows and m columns. We assume Flute [3] and Floram [2, 7] are executed without a trusted third party. Here Auth is Authenticity, GC-Comp is GC-Compatible.

Intuitively, we extract all garbling-related operations (with κ-overhead) to *outside* of stacking, leaving only (masked) truth tables operations subject to logarithmic stacking overhead. As mentioned, the construction is quite intricate; we present the high-level intuition of the approach in Section 4, and additional lower-level intuition in Section 5. Formal construction is in Section 6.

Performance. Concretely, in communication we improve over prior best garbled lookup as soon as $N \geq 2^{10}$, getting >5× reduction when $N \geq 2^{15}$ and >50× reduction when $N \geq 2^{20}$. Our protocol outperforms state-of-the-art GRAM [21] on our envisioned range of parameters – small to medium-sized databases, *e.g.*, with $N \leq 2^{21}$ rows, even in comparison to the *amortized* GRAM cost! We implemented our scheme and validate performance improvements. See Section 7 for details.

1.2 Applications

Privacy-Preserving Machine Learning (PPML) Look-up tables are widely used in PPML for computing non-linear functions in neural networks [30]. They are used to retrieve the coefficients for polynomial approximation [22, 31] or for direct output lookup. Sigma [8] uses size-2^{13} and 2^{16} LUTs to compute inverse and a size-2^{13} LUT for reciprocal square root. SecFloat [32] uses size-2^{12} LUTs for sigmoid and tanh. [33] shows LUTs can facilitate fixed-point comparison, which is essential in PPML [30, 29].

Client-Malicious Secure Inference. Our GC-based LUT can strengthen security in secure inference, an active sub-field in PPML. Secure inference considers a model owner with a proprietary model and a client wishing to run a sensitive query, while preserving privacy.

Muse [27] describes an attack, where a malicious client alters the output of a non-linear function to steal model parameters. Muse proposes a defense where all non-linear functions are computed using GCs with client being the evaluator. Thanks to GC authenticity, a malicious client cannot alter the computation results, thwarting the attacks. Muse only considers basic (comparison-based) activation functions. Our GC−LUT can be useful in secure inference with more advanced activation functions, *e.g.*, in transformer models.

2 Related Work

Private Information Retrieval PIR [5, 6] has communication sublinear in DB size N, while server computation is $O(N)$. [26] designed a PIR with communication $O(N^\epsilon)$ for an arbitrary ϵ; subsequent works achieved polylog communication. PIR enjoyed significant research attention, with many variants considered, including multi-server PIR. We are interested in GPIR, discussed next.

Garbled PIR GPIR is a garbled PIR lookup gate for use in GC. State-of-the-art GPIR is an immediate application of a more general work GCWise [9], which achieves sublinear communication and sublinear evaluator computation for 1-out-of-N evaluation of uniform-topology sub-circuits. [9] highlights the GPIR use case, where it achieves $\tilde{O}(\sqrt{N} \cdot (\kappa + m) \cdot \kappa)$ communication cost – here $\tilde{O}$ includes significant $\mathsf{polylog}(N)$ factors. Garbler's computation is $\tilde{O}(N(\kappa + m)c_\kappa)$, i.e., linear in N, while the evaluator has sublinear

Approach	**Communication (bits)**	**Computation (bit ops.)**
`Toss` (Ours)	$O(\sqrt{N}m\sqrt{\kappa})$	$O(Nm\kappa + (\sqrt{\frac{N}{\kappa}}m + N)c_\kappa)$
`logrow`	$(n-1)\kappa + nm\kappa + Nm$	$O(Nm\kappa + (N(1+\frac{m}{\kappa}) + nm)c_\kappa)$

Table 2: Comparison with `logrow` for $[\![a]\!] \mapsto [\![T[x]]\!]$ inside GC where $T : \{0,1\}^n \to \{0,1\}^m$ is a table with $N = 2^n$ rows. κ is the security parameter. c_κ is the computation cost of a hash evaluation.

computation $\tilde{O}(\sqrt{N}mc_\kappa)$. GCWise relies on heavy sampling machinery. [9] does not include performance evaluation, and does not claim practical efficiency. We view it as a feasibility result.

Garbled RAM GRAM [21, 20] allows oblivious access to an array in a garbled circuit with $O^{!}((\log N)m\kappa + (\log^3 N)\kappa)$ communication and computation per access. It requires an expensive initialization – $O(N)$ write accesses each with the above cost – making it unsuitable for infrequent lookups, where initialization cannot be amortized.

Garbled Lookup Table GC−LUT logrow [18] is the most relevant prior work, offering a practical implementation of GPIR. `logrow` allows GC lookups from a N-row m-column table with $O(m\kappa \log N + Nm)$ bits of communication, reducing the number of garbled rows in a LUT from linear to logarithmic in N. Although logrow works well for small N, the overall communication cost remains linear in N. Our scheme has lower communication cost for $N \geq 2^{10}$. In `logrow`, the LUT needs not be known to E, while our scheme requires it to be public.

We compare `logrow`'s costs to `Toss` in Table 2.

One-Hot Garbling (OHG) The above techniques support oblivious lookup that keeps the index private. If we allow evaluator E to learn the index, OHG [15] requires only $\log_2 N \cdot \kappa$ bits [11] of communication, still with $O(N)$ computation. As it reveals the index, OHG can only be securely used with specific well-structured functions (see [15]) and is not applicable to general lookup tables. OHG is an important building block in our scheme.

Stacked Garbling As suggested in [18], recursive stacking can be used to implement a lookup with $O(\log N(\log N + m)\kappa)$ bit of communication. This blows up the computational cost to $O(N^{2.389} \cdot m \cdot \kappa)$ [10]. This does not scale: lookup of a database with 2^{16} entries requires $> 2^{38}$ CCRH evaluations.

Interactive Lookup For completeness, we mention, but do not extensively discuss recent *interactive* 2PC lookup schemes. Interactive schemes incur communication latency costs, and are undesirable in many scenarios. Indeed, for example, VISA GC MPC [36] reports 1585× speed up over a prior multi-round GMW-style construction [35], of which $\approx 59\times$ improvement comes from eliminating latency (and only $\approx 27\times$ comes from other technical improvements).

Flute [3] is a recent interactive 2PC technique for oblivious lookup with total communication cost of $O(N + m)$ bits and $O(N)$ OTs during preprocessing. Its communication cost is slightly better than our protocol only when $N = o(m^2 \cdot \kappa)$. In other words, our *non-interactive* protocol is more communication-efficient for larger N. Floram [7] uses distributed point function to implement oblivious look up with $O(\kappa \log N)$ communication and $O(Nm + Nc_\kappa)$ computation. Floram requires $\log N$ rounds of communication in pre-processing. Crucially, it requires constant-round online communication for *each* invocation, making it incompatible with GC.

3 Preliminaries

3.1 Notation

For arrays and bit strings, we use 0-based indexing. Let $s[0]$ be the most significant bit of a bit string s or a binary number it represents.

- G is the circuit generator. We refer to G as he/him.
- E is the circuit evaluator. We refer to E as she/her.
- N denotes the number of rows in a LUT.
- $n = \lceil \log_2 N \rceil$ is the bit-width of the LUT row index.
- m denotes the number of output bits (LUT columns).
- B denotes the number of branches in stacked GC.
- α denotes the active branch's index.
- $b = \lceil \log_2 B \rceil$ is the number of bits needed to represent α.
- a is of the (maximum) number of input bits to each branch.
- κ denotes the computational security parameter (e.g. 128).
- $\mathsf{lsb}(x)$ denotes the least significant bit of x.
- $\langle\langle X^G, X^E \rangle\rangle$ denotes a pair of garbling shares, where X^G and X^E are held by G and E, respectively.
- $\Delta \in \{0,1\}^\kappa$ is a global Free XOR key known to G; $\mathsf{lsb}(\Delta) = 1$.
- $[\![x]\!]$ denotes garbled sharing of a value (bit) x. We use Free-XOR sharing $[\![x]\!] = \langle\langle X, X \oplus x\Delta \rangle\rangle$.
- $\langle A, B \rangle = \sum_i A[i] \cdot B[i]$ is the inner product of arrays A, B.
- w-label is the label corresponding to wire value w.
- c_κ denotes the computational cost of a CCRH evaluation.
- *GC material*, or just *material*, is a string encoding of the garbled gates of the circuit. E.g., in stacked GC, material includes gadgets material, as well as stacked branch material.
- $[a, b)$ denotes an array $(a, a+1, \ldots, b-1)$.
- $[a, b]$ denotes an array $(a, a+1, \ldots, b)$.
- $[b]$ is a shorthand for $[0, b)$.
- $A[a{:}b)$ denotes $(A[a], \ldots, A[b-1])$ for an array/string/table A. When x is an integer in a known domain, we treat x as a 0-padded bitstring. E.g., for $x = 6 = 0110_2 \in [16]$, $x[0{:}2) = 01_2$. $A[a{:}b]$ is defined similarly, with b-th item included.
- $\mathbb{1}\{S\}$ outputs 1 if S is true; otherwise, it outputs 0.
- We permute α-th truth table by linearly shifting it by γ_α, resulting in the permutation π_{γ_α}. π_{γ_α} is naturally defined both on scalar indices and entire tables.
- x is the lookup index and can be parsed as $\alpha || \beta$, where β is the lookup index in the α-th sub-table.

3.2 GC Definition

3.2.1 Circular Correlation Robustness Hash

Definition 1 (Circular Correlation Robustness, [37]). *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 $\mathcal{A}$:*

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

3.2.2 Garbling Scheme

Definition 2 (Garbling Scheme [1])**.** *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* $\mathcal{C}$*. It outputs GC material* $\mathcal{M}$ *and two strings* e, 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*, and outputs cleartext* $y \in \{0,1\}^m$*. It may also output* $\perp$ *to indicate failure.*

3.3 Stacked Garbling

We include formal definition of garbling schemes in in Section 3.2.2. We assume circular correlation robust hash (CCRH) H [4].

Stacked garbling (or Stacked GC, SGC) [24, 17, 13] is a GC technique that allows for efficient handling of programs with conditional branching. The technique allows the garbler to transmit GC material proportional only to one program execution path (e.g., the active branch), and not to the full circuit representing the program.

The key idea of SGC is that the garbler G garbles each of B branches C_i for $i \in [0, B)$ starting from random seeds. G thus obtains B strings, each representing a collection of garbled gates. These strings (and in general, any serialized representations of collections of garbled gates) are called material. G then *stacks* (XORs) these B materials, yielding material of the entire conditional, which is proportional to the longest branch. G sends this whole material, as well as the seeds of the inactive branches, to E. If E knew the active branch, she could reconstruct the inactive branches and *unstack*, or XOR-out, the active branch, which then she can evaluate.

Of course, E does not know which branch is active, so she makes B guesses, exactly one of which must be correct, and obtains B sets of output labels. Notice, there are B^2 ways for the evaluation to proceed: for each combination of (true active branch, guessed active branch), there will be a distinct evaluation by E, resulting in a distinct set of output labels.

The clever punchline of [13] is that G *can predict* all possible output labels, including garbage labels. Valid labels are known to G because he is the garbler. To predict garbage labels, G arranges the circuit so that all inputs to inactive branches are set to 0, thus making evaluation by E fixed (deterministic) for each incorrect guess, and future actions of E replayable by G at garbling time. Then G can prepare the output garbling gadget, called out-mux, which assembles labels of all branches and generates output labels. The setup preceding the branches – routing the inputs to the right branch and setting inactive branches inputs to 0 – is in-mux. Seeds of the inactive branches are delivered to E via a mechanism separate from in-mux.

Communication Cost of Stacking There are four sources of the communication cost: in-mux (and inactive branch seed distribution system), out-mux, and the stacked material. Clearly, the cost for the stacked material is that of the longest branch. Seed distribution, in-mux and out-mux costs are optimized in [17, 13] for their use case where B and number of branch inputs are relatively small and branch size are relatively large. However, when there are many small branches with many inputs, the case in our GPIR stacking, they become bottleneck, negating the benefit of stacking. Thus, reducing in-mux (with seed distribution) and out-mux costs is crucial for GPIR.

A natural implementation of in-mux with seed distribution (via standard encrypted truth tables), costs $O(B^2 a\kappa)$ communication, where a is (maximum) branch input size. Crucially, this can be improved to $O(Ba\kappa)$ using a gadget we call *private one-hot garbling.* To our knowledge, this gadget is not described or constructed in the literature, however we believe a variant is implemented in Heath *et al.* [17]'s codebase. We thus don't take credit for its design, but do use it and present an explicit construction in Section 5.1 and Appendix B.

out-mux gadget can be implemented with $2 \cdot B \cdot m \cdot \kappa$ bits, where m is the number of branch outputs. Each output wire label is encrypted with all B keys (including $B-1$ dummy keys) that E would obtain for that wire in the course of evaluation, thus hiding active branch index from both G and E .

In sum, the total communication cost of stacking can be reduced to $O(B(a+m)\kappa + |C|\kappa)$, where $|C|$ is the size in gates of the longest branch.

3.4 LogStack

Stacking GC *increases* computation, making it a bottleneck even for modest B, as low as $B = 2$ or 3 for the original SGC for common scenarios [16]. As discussed above, there are B^2 combinations of active/guessed branches. Each needs to be evaluated, implying computational cost $O(B^2)$ of branch garblings. LogStack [17] cleverly reduces this cost to $O(B \log B)$ with an observation that many unstacking materials and predicted labels can be reused at both garbling and evaluation time. Still, LogStack's logarithmic computation overhead implies the corresponding overhead of stacked `logrow` over plain `logrow` (*cf.* Section 1.1), significantly *increasing* total runtime while reducing communication. One of our main contributions is removing this overhead.

As we build on the ideas of LogStack, we explain its techniques in detail. At the high level, LogStack derives the seeds for stacking and unstacking materials using *SortingHat*, a custom seed distribution gadget: The parties secret-share the seeds in the nodes of a binary tree (the SortingHat) over B leaves, and then expand them to get the actual seeds for garbling. Details follow:

3.4.1 The SortingHat garbled gadget

SortingHat takes as input $[\![\alpha]\!]$, the garbled sharing of the active branch ID, and generates the SortingHat $\mathcal{SH}$, a secret sharing of seeds arranged in a binary tree with B leaves, each corresponding to a branch.

G's share of $\mathcal{SH}$ is $\mathcal{SH}^G$, where each node, both internal and leaf, has two κ-bit seeds $s^{\mathcal{G}}$ (good) and $s^{\mathcal{B}}$ (bad). As the name suggests, only the good seeds can be used to generate the correct branch garbling materials. We denote the seed at the i-th leftmost node in the leaf level (*i.e.*, level $b = \lceil \log_2 B \rceil$) by $s_{i[0:b)}$. Here $i[0{:}b)$ denotes bits $i[0]||\dots||i[b{-}1]$ of the binary representation of i. We index the seeds of the inner nodes of the tree by correspondingly fewer bits. For example, the parent of the i-th leaf node will have seed $s_{i[0:b-1)}$. Such indexing conveniently aligns the seeds on each path wrt the binary representation of i: the index of a parent is always a prefix of the index of a child.

Each node's good seed is generated simply by hashing its parent's seed with a nonce v_i: seed s_i at level $\ell \in [1, b]$ is set to be

$$s_i^{\mathcal{G}} = H(v_i, s_{i[0:\ell-1)}^{\mathcal{G}})$$

All nonces v_i can be generated by G (e.g. via a global counter), and are revealed to E. Bad seeds are pseudo-randomly generated and are computationally indistinguishable from the good seeds.

In E's share $\mathcal{SH}^E$, each node has only one seed. The sibling roots of the α-th leaf (i.e., the sibling nodes of the α-th leaf's ancestors) will have the good seed; other nodes have bad seeds. That is, the seeds in α's sibling roots are $\{s_{\alpha[0:\ell)\oplus 1}\}_{\ell\in[1,b]}$. To illustrate, Figure 6 highlights the positions of sibling roots of $\alpha = 2$ (denoted by $\mathcal{G}$). By *sibling subtrees* we mean subtrees under α's sibling roots.

3.4.2 E's unstacking using $\mathcal{SH}$

Recall, when E guesses the active branch and wants to unstack the (stacked) material $\mathcal{M}$ for this branch, E uses the seeds for the guessed-inactive branches to derive the corresponding *unstacking materials*:

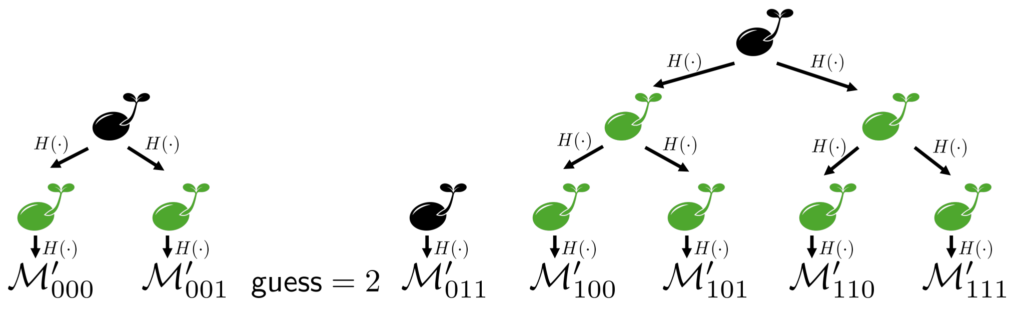

Figure 1: Unstacking illustration. Suppose E guesses branch $i = 2$ is active. She identifies the sibling roots (shown in black) and re-derives the seeds (green) in their subtrees. Then, E uses the seeds at the leaves to generate the garbling material for the guessed-inactive branches. If the guessed i was correct, all these seeds are good, and E correctly unstacks.

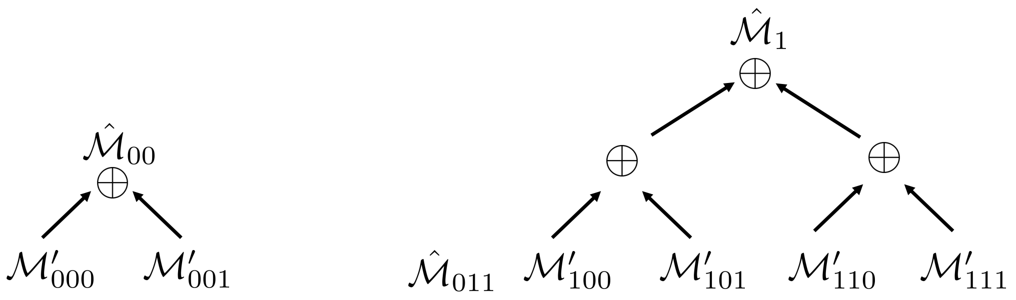

Figure 2: E's Computation in LogStack. Suppose E has expanded the seeds s_{00}, s_{011}, s_1 and generated materials $\{\mathcal{M}'_j\}_{j\in[8]\setminus\{2\}}$. E XORs these materials in each subtree to compute aggregate materials $\hat{\mathcal{M}}_{00}, \hat{\mathcal{M}}_{011}, \hat{\mathcal{M}}_1$. Now, $\hat{\mathcal{M}}_1$ can be used when E guesses $i \in \{0, 1, 2, 3\}$ is the active branch, because E would expand the same s_1 to get $\mathcal{M}'_{100}, \ldots, \mathcal{M}'_{111}$. Likewise, $\hat{\mathcal{M}}_{00}$ can be used for guess $\in \{2, 3\}$.

When E guesses the active branch index is i, E makes a copy of $\mathcal{SH}^E$, and uses the seeds in i's sibling roots in $\mathcal{SH}^E$ to re-derive all other seeds in i's sibling subtrees (see Figure 1 for illustration). Clearly SGC works because G can emulate E's actions, and, if $i = \alpha$, E will correctly unstack.

3.4.3 Logarithmic computational cost

It is easy to see the source of the computational savings: both G and E can reuse entire subtrees in their stacking and unstacking of materials – see Figure 2 for intuition and illustration.

The cost accounting indeed does follow the intuition: For a tree node at the j-th level, computational cost of garbling and stacking its leaf branches is $O(2^{b-j}|C|)$. (Here we assume all branches are of the same size $|C|$. We omit the c_κ factor associated with garbling circuit gates.) Since the j-level has 2^j nodes and there are b levels, the total cost is $\sum_{j\in[1,b]} 2^j \cdot O(2^{b-j}|C|) = O(B \log B \cdot |C|)$. When E unstacks for a guessed branch, it will select $\log_2 B$ nodes as sibling roots and then XORs the materials derived from them. Thanks to the precomputed subtree stacks, the computation is $O(\log B \cdot |C|)$. For all B guesses the total cost is $O(B \log B \cdot |C|)$.

G's computation follows the same asymptotics as E's, but with higher concrete costs: In addition to garbling and stacking the subtrees in $\mathcal{SH}$, G also evaluates $O(B \log B \cdot |C|)$ unstacked materials with dummy inputs to compute garbage labels. Looking ahead, this presents the main challenge for removing the logarithmic overhead on our stacked GPIR.

For completeness and ease of comparison with our protocols, we include formal LogStack in Appendix B.

- Parameters: Parties agree on the input size n, m.
- Input:
  - Parties input $[\![x]\!], [\![y]\!]$ where $x \in \{0,1\}^n, y \in \{0,1\}^m$.
  - E inputs x
- Output: Parties output a sharing $\langle\langle X^G, X^E \rangle\rangle = [\![\mathcal{H}(x)^E \otimes y]\!]$ such that for each $i \in [2^n], j \in [m]$

$$X^E[i][j] = \begin{cases} X^G[i][j] \oplus y[j] \cdot \Delta, & \text{if } i = x; \\ X^G[i][j] & \text{otherwise} \end{cases}$$

- Communication : G sends to E $(n + m - 1)\kappa$ bits.
- Computation : The parties use $O(2^n m c_\kappa)$ computation.
- Procedure:
  - Parties compute $\langle\langle W^G, W^E \rangle\rangle = [\![\mathcal{H}(x)]\!]$ via PubOneHot.
  - G computes Z^G by setting for all $i \in [2^n], j \in [m]$.

$$Z^G[i][j] = H(v_{i,j}, W^G[i])$$

  - G then sends to E for all $j \in [m]$.

$$C_j = \bigoplus_{i \in [2^n]} Z^G[i][j] \oplus Y^G[j]$$

  - E then computes

$$Z^E[i][j] = \begin{cases} C_j \oplus \bigoplus_{k \neq x, k \in [2^n]} Z^E[k][j] \oplus Y^E[j], & \text{if } i = x; \\ H(v_{i,j}, W^G[i]), & \text{otherwise.} \end{cases}$$

Figure 3: Outer Product Garbling OuterProduct. It requires revealing x, one of the inputs, to E.

To conclude, LogStack is a garbling scheme with communication

$$O(|C| + |\mathcal{SH}_B| + |\textsf{in-mux}_{B,a}| + |\textsf{out-mux}_{B,m}|)$$

and computation

$$O(B \cdot \log B \cdot |C| + |\mathcal{SH}_B| + |\textsf{in-mux}_{B,a}| + |\textsf{out-mux}_{B,m}|)$$

Here, $|\mathcal{SH}|$, $|\textsf{in-mux}|$, and $|\textsf{out-mux}|$ are the size of Sorting Hat, in-mux, and out-mux, parameterized by the number of branches B, input size a, and output size m, respectively.

The exact cost of Sorting Hat, in-mux, and out-mux will be defined in Section 5.1. Notice that reducing the size of them can immediately reduce the cost of stacking.

3.5 One-Hot Garbling

Definition 3 (One-Hot Encoding). *One-hot encoding $\mathcal{H}(x)$ of $x \in \{0,1\}^n$ is a $(0,1)$-vector of length 2^n with 1 in position x and zeros in all other positions. That is,*

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

- Parameters: The input size n and output size m.
- Input:
  - G inputs a 2^n-row m-column truth table T.
  - Parties input a shared index $[\![\beta]\!]$ where $\beta \in \{0,1\}^n$.
- Output: length-m sharing $\langle\!\langle Y_G, Y_E \rangle\!\rangle = [\![T[\beta]]\!]$ to parties
- Communication : $(n-1)\kappa + nm\kappa + 2^n m$ bits.
- Computation : Each party uses $O(2^n m c_\kappa)$ computation.

Figure 4: The interface and the cost of `logrow` [18].

[15] proposed efficient One-Hot Garbling (OHG), a protocol that produces the garbled labels $\mathcal{H}(x)$, but requires that E learns x. Authors show that OHG is a convenient GC gadget: it allows to efficiently look up and enter into the garbled circuit a value e.g., from a function truth table. Note, if the table is public, the lookup is communication-free once OHG is evaluated, thanks to Free XOR [23, 25]. Despite leaking x to E, the authors show that OHG is useful in MPC of certain homomorphic functions: a random mask is applied to the private input, the function value is computed on the masked input by looking up a truth table, and then, inside GC, the (appropriately modified) mask is taken off the result to obtain the correct output. Due to the space limit, we provide the formal description of one-hot garbling in Figure 14 in the Appendix.

Our GPIR construction uses an OHG's extension OuterProduct, which is also proposed in [15]. It outputs the the outer product of an (public) one-hot vector and another vector. We provide the interface in Figure 3 and briefly explain the construction in Appendix A.

`Toss` uses OHG [15] (as part of OuterProduct), and a fully private, but less efficient, OHG (see Section 5.1).

3.6 GC−LUT `logrow`

GC−LUT takes as input a lookup table F as a $2^n \times m$ bit matrix that specifies a function $f : \{0,1\}^n \to \{0,1\}^m$ and a garbled sharing of index $[\![a]\!] \in \{0,1\}^n$, and outputs a garbled sharing of the looked-up result $[\![f(a)]\!]$.

Basic GC of course can be used to implement the size-2^n garbled table. Until recent work [18], the only (tiny) improvement over basic GC was the garbled row reduction [28], which shaves a single garbled row off any garbled table.

3.6.1 `logrow`

[18] drastically reduces the communication cost to $\approx n(m+1)\kappa + 2^n m$ bits. We provide the interface of `logrow` and its exact costs in Figure 4.

`logrow` achieves factor κ asymptotic improvement over the basic GC implementation of GC−LUT. Inside, it "upgrades" OHG to support garbled lookup without revealing the index β to E. At the high-level, G masks the index β by a random value $\gamma \in \{0,1\}^n$, setting $a = \beta \oplus \gamma$. G then permutes (the rows in) f according to γ and masks the permuted f by a random function $r : \{0,1\}^n \to \{0,1\}^m$. Namely, G constructs a masked function $f'(a) = f(a \oplus \gamma) \oplus r(a)$. After G sends the lookup table of f' to E, parties can jointly lookup $[\![f'(a)]\!]$ via OHG. A clever trick in their construction for achieving concrete efficiency, `logrow` recursively evaluates $[\![r(a)]\!]$ by evaluating n "half-hidden" random functions, each of which cost $m \cdot \kappa$ bits of communication, totaling $n \cdot m \cdot \kappa$ bits. Finally, the parties can compute

$$[\![f'(a)]\!] \oplus [\![r(a)]\!] = [\![f((\beta \oplus \gamma) \oplus \gamma) \oplus r(a) \oplus r(a)]\!] = [\![f(\beta)]\!]$$

The total communication cost is $((n-1) + nm)\kappa + 2^n m$ bits because

- OHG costs $(n-1)\kappa$ bits
- The n half-hidden random function costs $nm\kappa$ bits.
- The (plaintext) masked truth table costs $2^n m$ bits

Looking ahead, our reduction in communication cost mainly stems from shrinking the masked truth table sent by G, the dominant term in `logrow`'s cost as N increases.

4 High-Level Overview

We start with outlining our basic idea in Section 4.1. Next, in Section 4.2, we outline the more challenging obstacle: the increased computation cost. In Section 4.3 we explain at the high level our solution. Intuition of several lower-level aspects of the construction is presented in Section 5.

4.1 Basic Idea: Stacking GC−LUTs

To reduce the communication cost of GPIR, we start with a simple and powerful idea: splitting the large LUT into several chunks and stacking them in GC.

In more detail, parties split the N-row m-column table T into $B = 2^b$ sub-tables $\{T_i\}_{i\in[B]}$ where each sub-table consists of the i-th chunk of consecutive rows in T *i.e.*, $T_i[j] = T[i||j]$ for $j \in [N/B]$. Since the sought row is in precisely one sub-table, all others are inactive, and the sub-table garbled materials can be stacked to save communication. Specifically, let the lookup index be $x = \alpha||\beta$, then $\alpha \in [B]$ is the active branch index and β is the lookup index in T_α. Clearly, $T[x] = T_\alpha[\beta]$.

Recall, for a N'-row m-column table, `logrow` requires transmitting $\approx \lceil \log_2 N' \rceil \cdot (m+1) \cdot \kappa + N' \cdot m$ bits [19]. Our hope is that by setting $B = \sqrt{N/\kappa}$[1], we can obtain a GPIR protocol with communication $O(\sqrt{N} \cdot m \cdot \sqrt{\kappa})$.

This hope for an easy solution is quickly crushed: simply stacking `logrow` incurs logarithmic factor overhead in computation, making computation the bottleneck instead and actually *decreasing* overall performance on our target setup (laptop+WAN/LAN). Additionally, even the communication is not improved: the savings of stacking are offset by the large in-mux gadget.

4.2 Obstacle: LogStack's Overheads

While applying the basic SGC and LogStack can reduce communication, it *increases* computation, causing computation to become the bottleneck. (See Section 3.4.)

For a N'-row m-column table, `logrow`'s computational cost mainly comes from the $O(N')$ CCRH evaluations and $O(N'm)$ XOR operations on κ-bit labels, totaling to $O(N'c_\kappa + N'm\kappa)$, where c_κ is the computational cost of CCRH evaluation [18].

In our application, we break the large LUT into $B = \sqrt{N/\kappa}$ chunks, so in our case $N' = N/B$. The total computation of implementing our LUT directly with LogStack is $O(B \log B|C|) = O(B \log B \cdot N'(c_\kappa + m\kappa)) = O(\log(\frac{N}{\kappa})N(c_\kappa + m\kappa))$.

Even for medium-size databases, it implies an order of magnitude computational penalty (*e.g.*, 8.5× for $N = 2^{24}$) compared to unstacked `logrow`.

We also must improve LogStack's communication cost due to its components SortingHat which costs $O(B^2\kappa)$[2], and in-mux (costing $O(B^2 \log_2 N' \cdot \kappa)$, cf Section 5.3).

4.3 Intuition of our Computationally-Efficient Solution, `Toss`

Our core low-level technical contribution is communication-efficient stacked GPIR that avoids LogStack-introduced factor $O(\log B)$ computational overhead on garbling branch-size material. (That is, we do not have the term $O(N \cdot \log B \cdot \kappa)$ in our computational cost). We seek to reduce the number of expensive CCRH

[1] Setting $B = \sqrt{N}$ is the more natural first thought. Unfortunately, because the communication cost of our in-mux is $O(Bm\kappa)$, setting $B = \sqrt{N}$ yields suboptimal total GPIR communication $O(\sqrt{N}m\kappa)$ bits. We discuss the costs of our in-mux in Section 5.3.

[2] SortingHat is implemented by G sending a B-by-$(B-2)$ encryption table of seeds, causing $O(B^2\cdot\kappa)$ communication overhead. It is undesirable because it will lead to $O(N)$ communication cost in our application.

evaluations and the corresponding cost of managing (reading to and writing from RAM) of κ-bit labels, while keeping LogStack's reduction in communication.

Our protocol, `Toss` (table-only stacking of sub-tables), has computation cost $O(Nm\kappa + (\sqrt{\frac{N}{\kappa}}m + N)c_\kappa)$. For our intended use cases where $N = \omega(m^2/\kappa)$, our computation becomes $O(N(c_\kappa + m\kappa))$, similar to no-stacking computation.

We now introduce table-only stacking and explicate some details. As we go, we encounter technical issues and describe solutions. It is instructive to start with E's part of the computation, and then address the more challenging issue of G's work on garbage collection.

As the name "table-only stacking" implies, the technical core of our approach is to stack (LogStack-style, incurring logarithmic in B overhead) only the essential part of GC−LUT – the *plaintext* LUT – but not the full garbled materials of the sub-tables. We exploit the structure of the `logrow` sub-circuits to extract the (expensive) garbling operations from inside stacking, where they would have been run $B \log B$ times. Instead, we apply them outside stacking, executing them only B times. As a result, only plaintext operations on LUTs are executed $B \log B$ times.

This means that, while stacking logarithmic overhead is required, it 1) *only* applies to computing on bits, rather than labels, and 2) does not apply to CCRH evaluations. In other words, we don't have terms $N \log B \cdot \kappa$ in our computation cost. In particular, the stacked materials for GC−LUT are reduced from XOR sum of all garbling materials (which contains κ-bit labels) to XOR sum of truth tables (without the κ-blowup).

4.3.1 E's table-only stacking

We first show how to eliminate E's $O(\log B)$ factor computational overhead of unstacking garbled material, which includes expensive regarbling of $B \log B$ branches.

Recall, in `logrow`, G needs to generate three pieces of material: 1) a masked and permuted truth table 2) labels to allow E to evaluate a uniformly random function to unmask the truth table, and 3) labels for one-hot garbling. Note, while communication costs of (2) and (3) are logarithmic in table size, they incur linear computation.

In our construction, we extract to outside of stacking the masking of item (1) above and garbling related to items (2) and (3). We run them only once for the entire GC−LUT computation. This achieves our stated goal for E of sublinear in N communication, and avoiding $O(\log B)$ computation overhead.

In more detail, G permutes[3] the (rows of the) sub-tables and XORs them, masks the resulting stacked table and sends it to E. In contrast with traditional stacking, our branch material stacking and unstacking does not require generation of garbled labels and consist of much cheaper operations. A positive side effect of avoiding stacking garbled material is the fact that we can use the same offset Δ in *all* branches. This is useful in designing an efficient `in-mux` gadget and G's garbage collection.

Of course, since we evaluate OHG outside of stacking, each branch is not a complete circuit, and we need a new design of `in-mux` to feed one-hot representation of permuted sub-table index $\pi_{\gamma_\alpha}(\beta)$ used in the `logrow` lookup. This means that the input to each sub-table, the one-hot encoding of the index β, is large. Routing this via GC gadgets naïvely would cost unaffordable $O(N\kappa)$ communication. Instead, we design an OHG gadget that delivers the correct one-hot vector for the correctly guessed branch, and an all-zero vector for incorrectly guessed ones. See Section 5.3. Its communication cost is $O((B + |\beta|)\kappa)$.

4.3.2 Improved G's garbage collection

Digging deeper, we notice that the above is insufficient to achieve the same asymptotics on G's computation. This is because during E's unstacking, the true branch index α is fixed, and E computes candidate truth tables (without incurring κ overhead) $B \log B$ times, and performs inner product over the truth tables and OHG (incurring κ overhead) only B times.

[3]Distinct random linear shift is used for each sub-table.

In contrast, to generate garbage-collection labels, G has to emulate E's actions (crucially, including computing the inner product of the truth tables and OHG, which incurs κ-overhead) on every possible α. Using LogStack's SortingHat trick, this can be done with $\log B$ overhead, but over κ-overhead operations. (Note, in LogStack this asymptotic imbalance between E and G computation does not arise because E performs entire stacking computation with κ-overhead, and has $B \log B \cdot \kappa$ complexity.)

Hence, the goal of our improved G's garbage-label generation is to achieve more computationally efficient inner product over the truth tables and OHG, which avoids $N \log B \cdot \kappa$ term in G's computation.

Our core observation here is that the calls to inner product can be reduced because of its linearity: Given an OHG, and two truth tables, the following is the same: 1) compute the two inner products and then XOR the resultant labels and 2) XOR two tables and then compute an inner product. Of course, (2) is much cheaper because κ-bit labels are touched only once.

We use this principle in computing the stacks corresponding to the subtrees in LogStack's SortingHat tree $\mathcal{SH}$, as follows: for a given node (good or bad), we generate the leaf (masked and permuted) truth tables, but apply the OHG inner product only to their XOR, and not to them individually.

This helps, but does not quite bring us to the desired asymptotics because for each possible true index α, there are logarithmic number of such stacks (corresponding to the sibling subtrees of α). These stacks, after the inner product computation, are represented in garbled form, and standard LogStack garbage computation incurs log-factor overhead over *garbled* values, which we wish to avoid.

We resolve this by changing how E evaluates $\mathsf{out\text{-}mux}$. Instead of asking G to essentially generate all possible garbage labels (as is done by both Stack and LogStack), we exploit the linearity of XOR and of the inner product in our table-only stacking, as well as the fact that inputs to inactive branches are real 0-labels.[4] This means that for the (masked) global LUT output y, and for the active branch index α, the XOR of all output labels is an α-dependent label encoding y. There are only B such labels, one for each α. This allows G to generate the $\mathsf{out\text{-}mux}$ efficiently. See Section 5.5 for details.

This completes the high-level description of our approach. We next (Section 5) discuss several aspects of our construction in more technical detail.

5 Lower-level Intuition

While Section 4 aims to explain the roadmap and the whole construction at the high level, in this section, we convey lower-level intuition and details of some important aspects of the construction. This section should be read after Section 4. The full `Toss` construction is formally presented in Section 6. We start, in Section 5.1, with explaining building efficient private OHG and $\mathsf{SortingHat}$, primitives used e.g., in Section 5.3.

5.1 $\mathsf{SortingHat}$ from Private One-hot Garbling

$\mathsf{SortingHat}$ construction of [17] is not optimized for the case of large B and small $|C|$, and using it as-is will prevent us from achieving the asymptotics we desire. Here we design an efficient private one-hot garbling and explain how we can build from it an efficient generically-applicable $\mathsf{SortingHat}$, used in our GPIR scheme. Additionally we use private OHG for $\mathsf{in\text{-}mux}$ in Section 5.3.

5.1.1 Private One-Hot Garbling

It is not clear how to adapt one-hot garbling of [15] to $\mathsf{SortingHat}$, since we must not reveal x to E, and our function is not of special homomorphic form. Rather, we need a *private* one-hot garbling, producing the garbled shares of one-hot encoding without leaking x to E. More precisely, we need a gadget to transform $[\![x]\!] \mapsto [\![\mathcal{H}(x)]\!]$, which we use in Section 5.1.2 to build $\mathsf{SortingHat}$.

[4]Revealing these 0-labels does not break security because in table-only stacking, stacked branches do not carry garbled material and hence garbled material is not reconstructed from seed during unstacking. These 0-labels do not help E to distinguish good and bad seeds.

- PARAMETERS: Parties agree on the input size n.
- INPUT: Parties input a shared index $[\![x]\!]$ where $x \in \{0,1\}^n$.
- OUTPUT: Parties output a sharing $\langle\!\langle X_G, X_E \rangle\!\rangle = [\![\mathcal{H}(x)]\!]$ such that for each index i:

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

- COMMUNICATION : G sends to E $(2^n - 2)\kappa$ bits.
- COMPUTATION : Each party uses $O(2^n c_\kappa)$ computation.
- PROCEDURE:
  - If $n = 1$: Parties compute and output $(1 \oplus [\![x]\!], [\![x]\!])$
  - Parties compute $[\![\mathcal{H}(x[0{:}n{-}2])]\!] = \mathsf{PrivOneHot}([\![x[0{:}n{-}2]]\!])$
  - Parties compute $[\![\mathbf{s}]\!]$ where $[\![\mathbf{s}]\!][i] = [\![\mathcal{H}(x[0{:}n{-}2)]\!][i] \wedge [\![x[n{-}1]]\!]$ for $i \in [2^{n-1}]$
  - Parties compute and output

$$\begin{aligned}
[\![\mathcal{H}(x)]\!][i] &\triangleq \begin{cases} [\![\mathcal{H}(x[0{:}n{-}2]]\!][i/2] \oplus [\![\mathbf{s}[i/2]]\!] & \text{if } i \text{ is even} \\ [\![\mathbf{s}[\frac{i-1}{2}]]\!] & \text{if } i \text{ is odd} \end{cases} \\
&= \begin{cases} [\![\mathcal{H}(x[0{:}n{-}2])[i/2]]\!] \wedge (1 \oplus [\![x[n{-}1]]\!]) & \text{if } i \text{ is even} \\ [\![\mathcal{H}(x[0{:}n{-}2])]\!][\frac{i-1}{2}] \wedge [\![x[n{-}1]]\!] & \text{if } i \text{ is odd} \end{cases} \\
&= \begin{cases} [\![1]\!], & \text{if } i \text{ is even} \wedge i/2 = x[0{:}n{-}2] \wedge x[n{-}1] = 0; \\ [\![1]\!], & \text{if } i \text{ is odd} \wedge \dfrac{i{-}1}{2} = x[0{:}n{-}2] \wedge x[n{-}1] = 1; \\ [\![0]\!], & \text{otherwise.} \end{cases} \\
&= \begin{cases} [\![1]\!], & \text{if } i = x; \\ [\![0]\!], & \text{otherwise.} \end{cases}
\end{aligned}$$

Figure 5: Private One-Hot Garbling $\mathsf{PrivOneHot}$.

This can be trivially implemented as a circuit, by using $(n-1)2^n$ AND gates to check if x matches the index of each entry, consuming $O(n2^n\kappa)$ bits of communication.

Instead, using a binary-tree structure, we reduce the cost by factor n, to $2^n - 1$ AND gates. We recursively define the procedure as follows: When $n = 1$, the parties simply set $[\![\mathcal{H}(x)]\!] = ([\![1 \oplus x[0]]\!], [\![x[0]]\!])$, where $[\![1 \oplus x[0]]\!]$ is computed by G injecting a constant value 1 into $[\![x[0]]\!]$, *i.e.*, $[\![1 \oplus x[0]]\!] = \langle\!\langle \Delta, 0 \rangle\!\rangle \oplus [\![x[0]]\!]$. For $n > 1$, assume that the parties have already computed $[\![\mathcal{H}(x[0{:}n-1))]\!]$, denoted by $[\![\mathbf{h}]\!]$. Then, the parties compute the garbling of the *shifting vector*:

$$\mathbf{s} \triangleq \begin{cases} \mathbf{h} & \text{if } x[n{-}1] = 1 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$[\![\mathbf{s}]\!]$ can be obtained by computing a conjunction of $[\![x[n{-}1]]\!]$ with each entry of $[\![\mathcal{H}(x[0{:}n-1))]\!]$ (total 2^{n-1} AND gates). Parties then compute:

$$[\![\mathcal{H}(x)]\!][i] = \begin{cases} \mathbf{h}[i/2] \oplus [\![\mathbf{s}[i/2]]\!] & \text{if } i \text{ is even} \\ [\![\mathbf{s}[(i{-}1)/2]]\!] & \text{if } i \text{ is odd} \end{cases}$$

It may be helpful to imagine the one-hot encoding $\mathbf{h}$ as the nodes of the second-last layer of a binary tree. If $x[n{-}1] = 0$, the computation above copies the value of the 1-entry to its left child; otherwise, it copies the 1-entry's value to its right child. It is similar to Figure 6's right figure. Figure 5, we list the complete protocol and show the correctness.

The total number of AND gates for mapping an n-bit x is $2^n/2 + 2^n/4 + \ldots + 2 = 2^n - 1$. (Note that this construction requires no AND gate for $n = 1$.)

5.1.2 SortingHat for Toss

The first and immediate benefit of the private one-hot garbling construction $[\![\mathcal{H}(\alpha)]\!]$ is reduction of the SortingHat's communication from $O(B^2\kappa)$ to $O(B\kappa)$. (See Section 4.2 for SortingHat [17] cost calculation.) At a very high level, our construction leverages the binary tree structure of SortingHat and uses $[\![\mathcal{H}(\alpha)]\!]$ to provide keys for decrypting the seeds at each node.

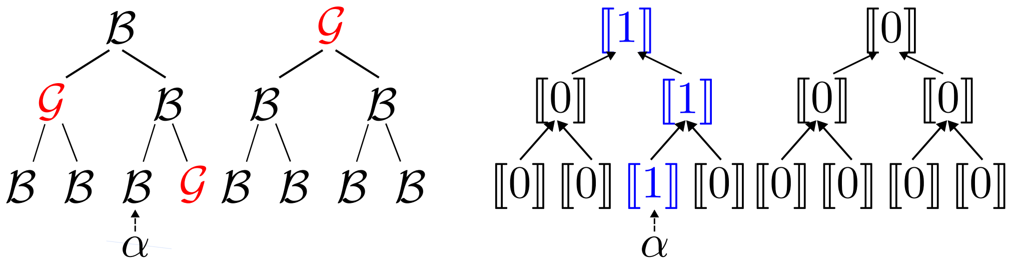

Figure 6: On the left is an illustration of E's SortingHat's output $\mathcal{SH}^E$: E should obtain good seeds (denoted by $\mathcal{G}$) only at sibling roots of the active branch (indexed by α) and obtain bad seeds (denoted by $\mathcal{B}$) everywhere else. On the right is the binary tree built by XORing $[\![\mathcal{H}(\alpha)]\!]$. Notice that good seeds are always in $[\![1]\!]$'s sibling nodes, enabling our efficient transmission for SortingHat.

As illustrated in Figure 6, our construction is based on the fact that all the good seeds are at α's sibling roots. With $[\![\mathcal{H}(\alpha)]\!]$, the parties can build a binary tree of garbled sharings where E has 1-labels only in the α-th leaf node and its ancestors. It can be done by setting $[\![\mathcal{H}(\alpha)]\!]$ as the sharing for the leaves and computing a node's sharing by XORing its children's shares, from the bottom to the top. Letting G use a node's sibling's 1-label as key to encrypt the good seed and 0-label for the bad seed, results in E obtaining good seeds just for the off-the-path sibling of α-th leaf node and bad seeds elsewhere.

This completes the high-level intuition of Toss's SortingHat; the formal construction is postponed to Section 6.1 (Figure 8).

5.1.3 in-mux for Generic Stacking.

While our main focus is table-only stacking used in Toss (see Section 5.3), we observe that the above idea is applicable generically to stacking. That is, we optimize in-mux by the improved private OHG. We postpone the presentation to Appendix B, but summarize the idea here: For the i-th branch, G uses an encrypted table to route the inputs, where the valid labels (corresponding to the inputs of the active branch) are encrypted by the 1-label of $[\![\mathcal{H}(x)[i]]\!]$ and the invalid labels (for inactive branches) are encrypted by the 0-label. Now, each branch requires only a $4a$-row encrypted table for input delivery. The total communication is reduced to $O(Ba\kappa)$ bits rather than $O(B^2a\kappa)$ bits. Altogether, the computation and communication cost of LogStack using our improved gadgets for SortingHat and in-mux are as follows:

Corollary 1. *Let $\mathcal{C}$ be a circuit containing B sub-circuits that each is of size S, and has input size a and output size m. Assume that XOR is free and a 2-input AND gate can be computed with $O(c_\kappa)$ bit operations with $\leq 2\kappa$ bits of communication. Then, LogStack is a garbling scheme with communication $O(S\kappa) + B(4 + 4a + 2m)\kappa$ and computation $O(B \cdot \log B \cdot S \cdot c_\kappa + B(a + m)c_\kappa)$.*

5.2 Stacking and Unstacking of Sub-Tables

In the following, we describe in more detail our ideas of the table-only stacking/unstacking (introduced in Section 4.3.1). Here, we focus on how to stack and unstack the sub-table for the active branch. The handling of inactive branches, in-mux, and out-mux will be discussed the next subsection.

The parties divide the database T into $B = 2^b$ consecutive sub-tables $T_0, \ldots, T_{B-1}$, each of $\frac{N}{B}$-row and m-column. We denote by $a = \log_2 \frac{N}{B}$ the index length to the sub-tables.[5]

Recall, we moved `logrow` and OHG circuits to outside of stacking; now only LUT data is stacked, and the seed for the i-th branch is only responsible for specifying permutation for the sub-table T_i.

During garbling, G uses the seed to derive an a-bit string γ specifying random linear shift. We define $\pi_\gamma(T)$ as a permutation over (the rows of) T such that $\pi_\gamma(T)[i] = T[i+\gamma]$. After deriving $\gamma_0, \ldots, \gamma_{B-1}$ from each branch's good seed and sampling a random bitstring $R \in \{0,1\}^{\frac{N}{B}\cdot m}$, G computes and sends to E the stacked material:

$$\mathcal{M} = R \oplus \pi_{\gamma_0}(T_0) \oplus \cdots \oplus \pi_{\gamma_{B-1}}(T_{B-1})$$

When E unstacks for the i-th branch, she obtains the seeds (from SortingHat), derives $\{\hat{\gamma}_j\}_{j\neq i}$, and computes $T'_\alpha = \mathcal{M} \oplus \bigoplus_{j\neq i} \pi_{\hat{\gamma}_j}(T_j)$. If E correctly guesses the active branch, she will reconstruct the correctly masked and permuted sub-table for branch $i = \alpha$, *i.e.*, she will get $T'_i = R \oplus \pi_{\gamma_\alpha}(T_\alpha)$, because all the seeds are good and give her $\hat{\gamma}_j = \gamma_j$ for all $j \neq \alpha$.

The parties then parse the index $x = \alpha || \beta$, where $\alpha \in \{0,1\}^b$ and $\beta \in \{0,1\}^a$. We assume that in-mux provides $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$ as input to the active branch. E's output of the active branch is

$$\begin{aligned}
&\langle T'_\alpha, [\![\mathcal{H}(\beta - \gamma_\alpha)]\!]\rangle \\
&= \langle R, [\![\mathcal{H}(\beta - \gamma_\alpha)]\!]\rangle \oplus \langle \pi_{\gamma_\alpha}(T_\alpha), [\![\mathcal{H}(\beta - \gamma_\alpha)]\!]\rangle \\
&= [\![R[\beta - \gamma_\alpha]]\!] \oplus [\![T_\alpha[\beta]]\!] \\
&= [\![R[\beta - \gamma_\alpha]]\!] \oplus [\![T[x]]\!]
\end{aligned}$$

The last step is to unmask the result:

$$[\![T[x]]\!] = [\![T'(\beta - \gamma_\alpha)]\!] \oplus [\![R[\beta - \gamma_\alpha]]\!]$$

Getting $[\![R[\beta - \gamma_\alpha]]\!]$ for Unmasking G and E simply look up $[\![R[\beta - \gamma_\alpha]]\!]$ using `logrow` implementing a LUT of size $(N/B)m$, costing inconsequential $a(m+1)\kappa + (N/B)m$ bits.

Further Reducing Communication Sending $\mathcal{M}$ and computing $[\![R[\beta - \gamma_\alpha]]\!]$ via `logrow` requires transmitting two masked tables, resulting in a communication cost of $2(N/B)m$ bits. This is in addition to the $a(m+1)\kappa$ bits already required for `logrow`.

To reduce this communication cost by $(N/B)m$ bits, we take inspiration from garbled row reduction and avoid sending $\mathcal{M}$ by setting $\mathcal{M} = 0\ldots00$, an all-zero string. To adjust for this $\mathcal{M}$, G sets

$$R = \bigoplus_{i\in[B]} \pi_{\gamma_i}(T_i),$$

Although R is no longer uniformly random, the security guarantees remain intact. Specifically, E's view is simulatable: $\mathcal{M}$ is a fixed material regardless of the inputs, and R is hidden from E thanks to the obliviousness guaranteed by `logrow`, which hides LUT from E.

5.3 Our in-mux for Toss

We present a communication-efficient implementation of in-mux for routing OHG vectors. The goal is to deliver $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$ to the active branch α, while ensuring that all inactive branches receive zero-hot vectors (i.e., vectors where all entries are 0-labels).

[5]Note, at the lowest level of detail, we are actually routing the OHG of the input (of size N/B) to branches. For the purpose of our discussion here, it is more instructive to consider the binary representation of the input of size $a = \log_2 \frac{N}{B}$.

5.3.1 Getting Input for the Active Branch

The first step is simple: G and E compute a sharing $[\![\beta - \gamma_\alpha]\!]$ in aid of GC−LUT, then expand it to an OHG $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$. Efficient OHG [15] reveals $\beta - \gamma_\alpha$ to E.

Specifically, G builds a look-up table with all values of γ derived from the good seeds in $\mathcal{SH}^G$. Using `logrow`, the parties look up $[\![\gamma_\alpha]\!]$, compute $[\![\beta - \gamma_\alpha]\!]$ (using an adder circuit), and subsequently expand it to $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$ using OHG.

5.3.2 Obliviously Distributing Inputs to All Branches

The next step is to route $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$ to the active branch α while distributing zero-hot vectors to all inactive branches. This can be framed as computing the outer product of $\mathcal{H}(\beta - \gamma_\alpha)$ and $\mathcal{H}(\alpha)$.

Recall, outer product $a \otimes b$ of vectors a and b is a matrix where the i-th row is $a \cdot b[i]$. For $\mathcal{H}(\beta - \gamma_\alpha) \otimes \mathcal{H}(\alpha)$: the i-th row is $\mathcal{H}(\beta - \gamma_\alpha)$ if $\mathcal{H}(\alpha)[i] = 1$; the i-th row is an all-zero vector if $\mathcal{H}(\alpha)[i] = 0$.

The i-th row is input to the i-th branch: the active branch receives $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$, and inactive branches receive garbled zero-hot vectors.

Given that $[\![\mathcal{H}(\alpha)]\!]$ is computed via private OHG (using the approach in Section 5.1.1), the garbled outer product $[\![\mathcal{H}(\beta - \gamma_\alpha) \otimes \mathcal{H}(\alpha)]\!]$ can be efficiently implemented using a variation of One-hot Garbling [15]. We call this construction $\mathsf{OuterProduct}$. For reference, the interface is provided in Figure 3, and more explanation can be found in Appendix A. This approach achieves a communication cost of $O((B + a)\kappa)$ bits.

5.4 E's Efficient Unstacking in `Toss`

E's unstacking in `Toss` follows LogStack: E derives permuted sub-tables from the recovered seeds and XORs them out. Details follow:

E pre-computes an unstacking table for every node in $\mathcal{SH}^E$. More precisely, we define $\hat{T}_{i[0:\ell)} \triangleq \mathsf{ExpPerm}(s_{i[0:\ell)})$ as *the aggregated sub-table* at node $i[0{:}\ell)$, computed by a local procedure $\mathsf{ExpPerm}$ (Figure 9), which uses LogStack ideas (cf Section 3.4.2, Figure 1 and Figure 2).

Then, when E wants to unstack for a guessed branch i, she computes the unstacked sub-table (recall, $\mathcal{M}$ is an all-zero string).

$$T'_i \triangleq \bigoplus_{j \in [1:b]} \hat{T}_{i[0:j) \oplus 1} \tag{1}$$

(Here, T'_i is implicitly parameterized by the active branch index α, unknown to E.) Crucially, this computation does not incur κ-factor.

Finally, E computes an inner product for each branch. Suppose that $\mathsf{in\text{-}mux}$ has provided to the parties $\langle\!\langle Y^G, Y^E \rangle\!\rangle = [\![\mathcal{H}(\alpha) \otimes \mathcal{H}(\beta - \gamma_\alpha)^E]\!]$. We define as the output label of the i-th branch

$$Z_i^E \triangleq \langle T'_i, Y^E[i] \rangle$$

5.4.1 Computation Cost Accounting for E's Unstacking

E performs pre-computation (costing $O(B \log B \cdot c_\kappa + Nm \log B)$) and inner product over the sub-tables and Y^E (costing $O(Nm\kappa)$). The term $O(B \log B \cdot c_\kappa)$ comes from expanding the seeds to derive γ_i, and $O(Nm \log B)$ is due to applying the permutation. The $\log B$-factor overhead is due to the LogStack-like procedures (cf Section 3.4.3).

Plugging in $B = \sqrt{N/\kappa}$, we get a ($\log B$-overhead-free) total computation cost $O(N(c_\kappa + m\kappa))$. The term $O(Nm \log B)$ from pre-computation cost is dominated by $O(Nm\kappa)$ for any practical N.

5.5 Garbage Collection

5.5.1 Our Custom out-mux

In Section 4.3.2 we highlighted the high cost of G directly generating all possible garbled labels. We design out-mux that does not require this. We exploit the fact that we have valid labels even in inactive branches.

Let E's output labels at the j-th branch be $Z^E_{\alpha,j}$, for active branch α. In our out-mux, E sums the branch output labels $\bigoplus_j Z^E_{\alpha,j} = (\bigoplus_j Z^G_{\alpha,j}) \oplus y\Delta$. To obtain the final output label, E combines (XORs) the above sum with the *translation* string U_α sent via a standard garbled table (E decrypts U_α using the labels of α).

5.5.2 Efficient Prediction for Translation

We thus reduced the garbage collection problem to G efficiently computing

$$U^G_\alpha \triangleq \bigoplus_{i\neq\alpha} Z^G_{\alpha,i} = \bigoplus_{i\neq\alpha} \langle T'_i, Y^G_i \rangle,$$

which allows G to prepare translation strings $U^G_\alpha \oplus V^G$ for each α to cancel out the α-dependent labels in inactive branches.

Consider Equation (1), which shows T'_i is the sum of the (stacked) sub-tables on x's sibling roots. These tables can be good (*i.e.*, $\hat{T}^{\mathcal{G}}_{x[0:j]}$) or bad (*i.e.*, $\hat{T}^{\mathcal{B}}_{x[0:j]}$), depending on the active branch index α. Recall, for a specific $\mathcal{SH}$ node, E obtains a good stacked table (derived from good seed) iff that node is one of α's sibling roots, *i.e.*,

$$\hat{T}_x = \begin{cases} \hat{T}^{\mathcal{G}}_x, & \text{if } x = \alpha[0{:}j) \oplus 1; \\ \hat{T}^{\mathcal{B}}_x, & \text{otherwise.} \end{cases}$$

for all $\mathcal{SH}^E$ node x on level j.

To save computation, G first pre-computes the inner product with all bad stacked tables, defining the baseline:

$$S_{\text{bad}} \triangleq \bigoplus_{k\in[B]} \langle \bigoplus_{j\in[1,b]} \hat{T}^{\mathcal{B}}_{k[0:j)\oplus 1}, Y^G_k \rangle$$

using $O(Nm\kappa)$ computation. We aim to leverage the linearity of $\langle \cdot, Y^G_x \rangle$ to cancel out $\hat{T}^{\mathcal{B}}_{x[0:j]}$ so that G can focus on computation related to the sub-tables on α's sibling roots, which will enable more efficient computation. More precisely, G only needs to compute the difference:

$$\begin{aligned} & S_{\text{bad}} \oplus \bigoplus_{x\neq\alpha} \langle T'_x, Y^G_x \rangle \\ &= \bigoplus_{x} \langle \hat{T}^{\mathcal{B}}_{x\oplus 1}, Y^G_x \rangle \oplus \bigoplus_{x\neq\alpha} \langle T'_x, Y^G_x \rangle \\ &= \bigoplus_{x\in[B]} \bigoplus_{j\in[1,b]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{x[0:j)\oplus 1}, Y^G_x \rangle \cdot \mathbb{1}\{x[0{:}j) = \alpha[0{:}j)\} \end{aligned}$$

where $\hat{T}^{\mathcal{G}\oplus\mathcal{B}}_x = \hat{T}^{\mathcal{G}}_x \oplus \hat{T}^{\mathcal{B}}_x$.

To simplify the computation, instead of computing many inner products then summing, G first sums the input labels, then computes fewer inner products.

Here, we define the aggregated labels for each subtree:

$$\hat{Y}^G_{\alpha[0:j)} \triangleq \bigoplus_{x\in[B]} Y^G_x \cdot \mathbb{1}\{x[0{:}j) = \alpha[0{:}j)\}$$

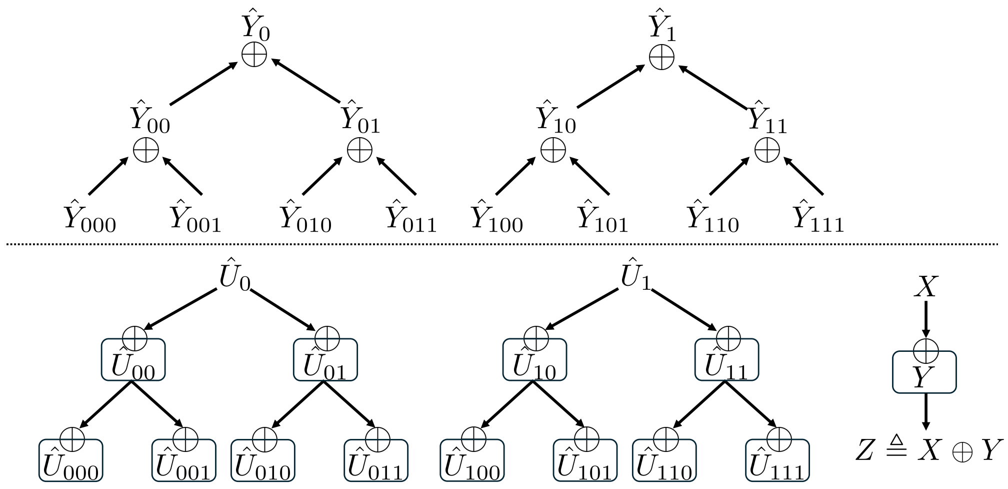

Figure 7: G's computation for translation without $O(\log B)$ computation overhead. As illustrated on the top, G prepares the summed input label for each internal nodes by XORing its children's labels. As illustrated at the bottom, G then computes $Z_i = \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{i\oplus 1}, \hat{Y}_i \rangle$ and finally computes the translation label for the branches (represented by the leaves) .

The following holds due to linearity.

$$\bigoplus_{x\in[B]} \bigoplus_{j\in[1,b]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{x[0:j)\oplus 1}, Y^G_x \rangle \cdot \mathbb{1}\{x[0{:}j) = \alpha[0{:}j)\} = \bigoplus_{j\in[1,b]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{\alpha[0:j)\oplus 1}, \hat{Y}^G_{\alpha[0:j)} \rangle$$

and the number of inner products is reduced from $O(B)$ to $O(\log B)$.

Computing all $\hat{Y}^G_{\alpha[0:j)}$ efficiently uses the tree structure, as illustrated in Figure 7. The first procedures is a **bottom-up aggregation**. G computes aggregated labels recursively: G starts with leaf labels Y^G_x. Then, each internal node's label is computed as the sum of its children's labels. The total computation cost is $O(Nm\kappa)$ because each label touched once.

The second procedures for to compute $\bigoplus_{j\in[1,b]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{\alpha[0:j)\oplus 1}, \hat{Y}^G_{\alpha[0:j)} \rangle$ efficiently for all $\alpha \in [B]$. G uses a **top-down propagation** approach: Instead of computing each branch independently: G compute inner products once at each tree node, where the node's aggregated label is inner-product with its sibling's difference table, *i.e.*, $Z_i = \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{i\oplus 1}, \hat{Y}_i \rangle$. Then, G Propagate results down the tree by XORing with parent values. Each leaf (branch) accumulates the sum of its ancestors' inner products Likewise, the total computation cost is $O(Nm\kappa)$ without $\log B$ factor.

Finally, G XORes each branch's leaf label with S_{bad} to get the translation label for the branch.

6 Formalization and Security Theorem

6.1 Formal Constructions of `Toss`

Our presentation focuses on `Toss`, efficient garbling of the PIR gate in GC. `Toss` is compatible with standard GC techniques. Thus we formulate our formal construction as a full garbling scheme for general circuits, which may now include PIR gates.

Construction 1 (GC-`Toss`)**.** `Toss` *is a garbling scheme (Definition 2) that supports the following gates:*

- *The standard two-input one-output XOR and AND gates*
- *LUT gates. A lookup gate is parameterized over a $2^n \times m$ truth table T known by G. It takes as input an index $x \in \{0,1\}^n$ and outputs $T[x] \in \{0,1\}^m$.*

- *PIR gates. A PIR gate is parameterized over a $2^n \times m$ database T known by both G and E and a number of branches. It takes as input an index $x \in \{0,1\}^n$ and outputs $T[x] \in \{0,1\}^m$.*

We define the garbling algorithms as follows:

- *$Gb(1^\kappa, \mathcal{C}) \to (\mathcal{M}, e, d)$:*
  - *It uniformly samples a global key $\Delta \leftarrow \{0,1\}^\kappa$ where the LSB of Δ is 1*
  - *It initializes e as an array. For each input wire $x[i]$ in $\mathcal{C}$, G uniformly samples a 0-label $X_w \in \{0,1\}^\kappa$ and sets $e[i] = (X_w, X_w \oplus \Delta)$.*
  - *It steps through the circuit gate-by-gate. It implements XOR gates by the Free XOR technique [25]. For AND gates, it uses the two-halve technique [37] (or three-halve [34]) and appends the ciphertexts into $\mathcal{M}$. For lookup gates, it uses the GLUT technique [19] and appends the generated material into $\mathcal{M}$. For GPIR gates, it runs the procedures defined in Figure 10, with the role of G and appends into $\mathcal{M}$ all messages sent by G to E.*
  - *It initializes d as an array. For each output wire $y[i]$, it retrieves the 0-label $Y[i]$ and sets*

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

  as a tuple. (Note that the nonce v varies with gates.) The LSBs are appended to ensure perfect correctness.
- *$En(e, x) \to X$: For each input bit $x[i]$, it outputs $e[i][x[i]] = (X[i], X[i] \oplus \Delta)$*
- *$Ev(\mathcal{M}, X) \to Y$: it steps through the circuit gate by gate, using $\mathcal{M}$ to map gate input labels to gate output labels: It handles XOR gates, AND gates, and Lookup gates using the evaluation procedures in Free XOR [25], Two-halve [37], and GLUT [19], respectively. For GPIR gates, it runs the procedures in Figure 10 taking the role of E.*
- *$De(d, Y) \to y$: For each output label $Y[i]$, it computes $Z[i] = H(v, Y[i])||\mathsf{lsb}(Y[i])$ and sets*

$$y[i] = \begin{cases} 0, & \text{if } Z[i] = d[i][0]; \\ 1, & \text{if } Z[i] = d[i][1]; \\ \perp, & \text{otherwise.} \end{cases}$$

We again highlight that GC-`Toss` is non-interactive, in the sense that all gates, including PIR gates, are garbled and sent to E simultaneously.

6.2 Security Theorems and Proofs

In this section, we formally state and prove our security claim.

We start with proving the correctness of `Toss`. Then, for the other security properties, we first prove that `Toss` is strongly stackable, which implies `Toss` is oblivious [17]. Then, any garbling scheme that is oblivious and has a decoding scheme for projective labels is also private and authentic [18].

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

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

Theorem 1. `Toss` *is correct.*

Proof. As `Toss` uses the implementation of AND, XOR, and GPIR gates from prior works [37, 25, 18], it inherits their correctness. We only need to prove the correctness of GPIR gates. We start with arguing the correctness of the sub-procedures invoked in Figure 10

- The correctness of private one-hot is shown in Figure 5.
- the correctness of SortingHat (Figure 8) is shown the proof for Lemma 1
- the correctness TableLogStk's is shown in the proof for Lemma 3

- PARAMETERS: The number of branches $B = 2^b$.
- INPUT: Parties input a shared (private) one-hot encoding $\langle\langle X^G, X^E \rangle\rangle = [\![\mathcal{H}(\alpha)]\!]$ where $\alpha \in \{0,1\}^n$.
- OUTPUT:
  - G receives an array $\mathcal{SH}^G = (s_i^{\mathcal{G}}, s_i^{\mathcal{B}})_{j\in[1,b], i\in[2^j]}$ where

$$s_i^{\mathcal{G}} \triangleq H(v_i^s, s_{i[0:j-1)}^{\mathcal{G}}) \text{ if } j > 1$$

  and for any $j \in [1,b], i \in [2^j]$,

$$s_i^{\mathcal{G}} \overset{c}{\approx} s_i^{\mathcal{B}}$$

  - E receives an array $\mathcal{SH}^E = (s_i)_{j\in[1,b], i\in[2^j]}$ where

$$s_i = \begin{cases} s_i^{\mathcal{G}}, & \text{if } i[0{:}j) = \alpha[0{:}j) \oplus 1; \\ s_i^{\mathcal{B}}, & \text{otherwise.} \end{cases}$$

- COMMUNICATION : G sends $(2B-2)\kappa$ bits to E.
- COMPUTATION : The parties use $O(Bc_\kappa)$ computation.
- PROCEDURES:
  - G samples $s_0^{\mathcal{G}}, s_1^{\mathcal{G}} \in \{0,1\}^\kappa$ and compute other $s^{\mathcal{G}}$ following the output requirement.
  - For $j \in [1,b], i \in [2^j]$, G computes

$$Y_i^G \triangleq \begin{cases} X^G[i], & \text{if } j = b; \\ Y_{i||0}^G \oplus Y_{i||1}^G, & \text{otherwise.} \end{cases}$$

  - For $j \in [1,b], i \in [2^j]$, G sends to E

$$C_i = s_i^{\mathcal{G}} \oplus H(v_i^C, Y_{i\oplus 1}^G \oplus \Delta)$$

  - For $j \in [1,b], i \in [2^j]$, G sets $s_i^{\mathcal{B}} = C_i \oplus H(v_i^C, Y_{i\oplus 1}^G)$
  - G outputs $\mathcal{SH}^G = (s_i^{\mathcal{G}}, s_i^{\mathcal{B}})_{j\in[1,b], i\in[2^j]}$
  - For $j \in [1,b], i \in [2^{j+1}]$, E computes

$$Y_i^E \triangleq \begin{cases} X^E[i], & \text{if } j = b; \\ Y_{i||0}^E \oplus Y_{i||1}^E, & \text{otherwise.} \end{cases}$$

  - For $j \in [1,b], i \in [2^j]$, E computes

$$s_i = C_i \oplus H(v_i^C, Y_{i\oplus 1}^E)$$

  - E outputs $\mathcal{SH}^E = (s_i)_{j\in[1,b], i\in[2^j]}$

Figure 8: Our Lightweight SortingHat. The interface is identical to LogStack [17]'s.

After confirming the correctness of each individual component, the overall correctness is easy to see in Figure 10. The only non-trivial part is why

$$[\![T[x]]\!] = [\![R[\beta - \gamma_\alpha]]\!] \oplus [\![\bigoplus_{i\in[B]\setminus\{\alpha\}} \pi_{\gamma_i}(T_i)[\beta - \gamma_\alpha]]\!]$$

which we have explained in Section 5.3 □

Definition 5 (Strong Stackability)**.** *A garbling scheme is* strongly stackable *[17] 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 2. `Toss` *is strongly stackable.*

Proof. Property (1): We prove that the input labels X and the material $\mathcal{M}$ are computationally indistinguishable from uniformly random strings.

For the input wire $x[i]$, $\mathcal{S}_{\mathsf{obv}}$ samples an uniform random string $\hat{X}[i] \in \{0,1\}^\alpha$ as the input label. All these labels are indistinguishable from the real label $X[i]$ or $X[i] \oplus \Delta$ because they are uniform strings.

Here, we show that all components in materials are indistinguishable from uniformly random strings:

- Parameters: Parties agree on a 2^n-row m-column public database T and number of branches $B = 2^b$
- Input: A party (G or E) inputs an index x and a seed s_x on level k of $\mathcal{SH}$. In other words, $|x| = k$
- Output: a 2^{n-b}-row m-column table $\hat{T}_x$
- Communication: No communication.
- Computation: $O(2^{b-k}c_\kappa + 2^{n-k}m)$
- Procedure:
  - Sets $s_x^{\mathsf{tmp}} = s_x$ and, for $j \in [1, b-k], i \in [2^j]$, computes
  $$s_{x||i}^{\mathsf{tmp}} \triangleq H(v_{x||i}^s, s_{x||i[0:j-1)}^{\mathsf{tmp}})$$
  - $\forall y \in \{0,1\}^b$ s.t. $y[0{:}k) = x$ for some k, defines a bitstring
  $$\gamma_y^{\mathsf{tmp}} \triangleq H(v_y^{\mathsf{tmp}}, s_y^{\mathsf{tmp}}) \in \{0,1\}^{n-b}$$
  and T_y^{tmp} be a 2^{n-b}-row m-col table s.t. $\forall j \in \{0,1\}^{n-b}$
  $$T_y^{\mathsf{tmp}}[j] \triangleq T[y||(j + \gamma_y^{\mathsf{tmp}})]$$
  - For $j \in [1, b-k], i \in [2^j]$, computes $y \triangleq x||i[0{:}j)$ and
  $$\hat{T}_y^{\mathsf{tmp}} \triangleq \texttt{if } j = b-k \texttt{ then } T_y^{\mathsf{tmp}} \texttt{ else } \hat{T}_{y||0}^{\mathsf{tmp}} \oplus \hat{T}_{y||1}^{\mathsf{tmp}}$$
  - Outputs T_x^{tmp}

Figure 9: ExpPerm: Local Procedure that expands a seed to a binary tree of randomness, permutes the chunked table with the generated randomness, and XOR-sums permuted tables.

- AND gates built from half-gate [37] are strongly stackable [17]. Thus, materials for AND gate are indistinguishable from uniformly random strings.
- Private One-Hot is built from AND and XOR gates, inheriting their indistinguishability.
- GC$-$LUT built from `logrow` is strongly stackable [18].
- SortingHat is indistinguishable because of Lemma 2
- $\beta - \gamma_\alpha$ is revealed to E in the material. Since γ_α is derived as a CCRH hash value, $\beta - \gamma_\alpha$ is uniformly masked.
- For the translation ciphertexts $C_0, \ldots, C_{B-1}$, they are encrypted by CCRH hash values, guranteeing their indistinguishability.

Property (2): `Toss` is also projective because every wire has two labels $X_0 \in \{0,1\}^\kappa$, and $X_1 = X_0 \oplus \Delta$ for the two wire values.

Property (3): one can use the LSB of the label as the color bit because $\mathsf{lsb}(\Delta) = 1$, ensuring the LSB of the two labels are different.

Property (4): we can simply define Key as a CCRH function with a fresh nonce. □

Definition 6 (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:*

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

where

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

Theorem 3. `Toss` *is oblivious.*

Proof. It immediately follows the fact that `Toss` is strongly stackable. □

Definition 7 (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 $(\mathcal{M}, e, d) \leftarrow Gb(1^\kappa, \mathcal{C})$, $X \leftarrow En(e, x)$, $y \leftarrow \mathcal{C}(x)$.

Definition 8 (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:*

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

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

Corollary 2. `Toss` *is private and authentic.*

Proof. Combining with the facts that `Toss` is oblivious and that `Toss` using projective (Free-XOR) labels, we can reuse the proofs in [18] (Appendix A and B) for `logrow`'s privacy and authenticity to show `Toss` is private and authentic. □

Theorem 4. *Assuming the existence of CCRH, GC-*`Toss` *(Construction 1) is a correct (Definition 4), oblivious (Definition 6), private (Definition 7), authentic (Definition 8), and strongly stackable (Definition 5) garbling scheme.*

- PARAMETERS: Parties agree on a 2^n-row m-column public database T and number of branches $B = 2^b$
- INPUT: Parties input a shared index $[\![x]\!]$ where $x \in \{0,1\}^n$.
- OUTPUT: Parties output a sharing $[\![T[x]]\!]$.
- COMMUNICATION : G sends $O((bn + (2^b + n)m)\kappa + 2^b(n-b) + 2^{n-b}m)$ bits to E.
- COMPUTATION : $O((2^n + b2^b)c_\kappa + Nm(b+\kappa))$
- PROCEDURE:
  - The parties parse $[\![x]\!] = [\![\alpha]\!]||[\![\beta]\!]$ where $\alpha \in \{0,1\}^b$ and $\beta \in \{0,1\}^{n-b}$
  - The parties compute a private one-hot share $[\![\mathcal{H}(\alpha)]\!]$ using Figure 5.
    $\triangleright$ `Communication:` $(2B-2)\kappa$. `Computation:` $O(Bc_\kappa)$.
  - G and E run SortingHat$([\![\mathcal{H}(\alpha)]\!])$ to obtain $\mathcal{SH}^G$ and $\mathcal{SH}^E$, respectively, using protocol of Section 5.1.2 and Figure 8.
    $\triangleright$ `Communication:` $(2B-2)\kappa$. `Computation:` $O(Bc_\kappa)$.
  - G extracts $(\gamma_1, \ldots, \gamma_B)$ from $\mathcal{SH}^G$ and populates them into a B-row $(n-b)$-col table Γ such that $\Gamma[i] = \gamma_i \ \forall i$
    $\triangleright$ `Communication: No. Computation:` $O(B(c_\kappa + n - b))$.
  - Parties compute $[\![\gamma_\alpha]\!] = \text{GC-LUT}(\Gamma, [\![\alpha]\!])$ using Figure 4
    $\triangleright$ `Comm.:` $b(n-b+1)\kappa + B(n-b)$. `Computation:` $O(B(n-b)c_\kappa)$.
  - Parties compute $[\![\beta - \gamma_\alpha]\!]$ via an adder circuit, and G reveals it to E.
    $\triangleright$ `Communication:` $4n\kappa$. `Computation:` $O(nc_\kappa)$.
  - The parties compute the following using Figure 3
    $$\langle\!\langle Y^G, Y^E \rangle\!\rangle = [\![\mathcal{H}(\beta - \gamma_\alpha) \otimes \mathcal{H}(\alpha)]\!] = \mathsf{OuterProduct}(\beta - \gamma_\alpha, \mathcal{H}(\alpha))$$
    $\triangleright$ `Communication:` $(n-b+B-1)\kappa$. `Computation:` $O(2^n c_\kappa)$.
  - Parties run TableLogStk (in Figure 11) with the following specification:
    * G inputs $(\mathcal{SH}^G, Z^G)$ and obtain $(\hat{U}_1^G, \ldots, \hat{U}_B^G)$. E inputs $(\mathcal{SH}^E, Z^E)$ and obtains U^E.
    $\triangleright$ `Communication: No. Computation:` $O((N + bB)c_\kappa + Nm\kappa + bNm)$
  - G samples V of same size as $\hat{U}_1^G$ and computes $C_i = V \oplus \hat{U}_i^G \ \forall i \in [B]$, and deliver C_α to E via garbled rows using $[\![\alpha]\!]$ as key.
    $\triangleright$ `Communication:` $Bm\kappa$. `Computation:` $O(Bmc_\kappa)$
  - Parties set
    $$[\![\bigoplus_{i \in [B]\setminus\{\alpha\}} \pi_{\gamma_i}(T_i)[\beta - \gamma_\alpha]]\!] = \langle\!\langle V, U^E \oplus (V \oplus U_\alpha^G \oplus A) \rangle\!\rangle$$
    $\triangleright$ `Communication: No. Computation:` $O(m\kappa)$
  - G computes a $2^{n-b} \times m$ table $R \triangleq \bigoplus_{i \in [B]} \pi_{\gamma_i}(T[i \mid\mid \cdot\,])$
    $\triangleright$ `Communication: No. Computation:` $O(Nm)$
  - Parties compute $[\![R[\beta - \gamma_\alpha]]\!] = \text{GC-LUT}(R, [\![\beta - \gamma_\alpha]\!])$ using Figure 4.
    $\triangleright$ `Communication:` $(n-b-1)(m+1)\kappa + 2^{n-b}m$. `Computation:` $O(2^{n-b}m\kappa)$
  - Parties outputs $[\![T[x]]\!] = [\![R[\beta - \gamma_\alpha]]\!] \oplus [\![\bigoplus_{i \in [B]\setminus\{\alpha\}} \pi_{\gamma_i}(T_i)[\beta - \gamma_\alpha]]\!]$
    $\triangleright$ `Communication: No. Computation:` $O(m\kappa)$

Figure 10: Our GPIR construction `Toss`. We count the cost in $\triangleright \ldots$, where the unit of communication cost is bits.

6.2.1 Security Proofs for Components

Here we prove the correctness of SortingHat and TableLogStk. They are more intricate than the proofs for the other components of `Toss`. The remaining proofs can be found in our full version.

Lemma 1. *Our SortingHat (Figure 8) produces correct outputs*

Proof. The correctness of G's output is self-evident by the procedure to compute $s^{\mathcal{G}}$. Now, we focus on E's output

$$\begin{aligned}
s_i &= C_{i[0:j)} \oplus H(v_i^C, Y_{i\oplus 1}^E) \\
&= s_i^{\mathcal{G}} \oplus H(v_i^C, Y_{i\oplus 1}^G \oplus \Delta) \oplus H(v_i^C, Y_{i\oplus 1}^E) \\
&= \begin{cases} s_i^{\mathcal{G}}, & \text{if } Y_{i\oplus 1}^G \oplus \Delta = Y_{i\oplus 1}^E; \\ s_i^{\mathcal{B}}, & \text{if } Y_{i\oplus 1}^G = Y_{i\oplus 1}^E. \end{cases}
\end{aligned}$$

because $s_i^{\mathcal{B}} = s_i^{\mathcal{G}} \oplus H(v_i^C, Y_{i\oplus 1}^G \oplus \Delta) \oplus H(v_i^C, Y_{i\oplus 1}^E)$.

From Figure 6, we can observe that

$$Y_{i\oplus 1}^E = \begin{cases} Y_{i\oplus 1}^G \oplus \Delta & \text{if } i = \alpha[0{:}j) \oplus 1; \\ Y_{i\oplus 1}^G, & \text{otherwise.} \end{cases}$$

Hence, E will obtain $s^{\mathcal{G}}$ precisely at the off-the-path sibling of the active branch α and $s^{\mathcal{B}}$ everywhere else, fulfilling the correctness requirement.

□

Lemma 2 (SortingHat Security). *Assuming the existence of CCRH, our SortingHat (Figure 8) reveals no information about the active branch α.*

Formally speaking*, there exists a simulator $\mathcal{S}_{\mathcal{SH}}$ such that*

$$\{C_i\}_{j\in[b], i\in[2^j]} \overset{c}{\approx} \mathcal{S}_{\mathcal{SH}}(1^\kappa, \mathbf{X}^E)$$

and for any node i:

$$s_i^{\mathcal{G}} \overset{c}{\approx} s_i^{\mathcal{B}}$$

Proof. For $j \in [1, b], i \in [2^j]$, $\mathcal{S}_{\mathcal{SH}}$ computes

$$Y_i^E \triangleq \begin{cases} X^E[i], & \text{if } j = b; \\ Y_{i||0}^E \oplus Y_{i||1}^E, & \text{otherwise.} \end{cases}$$

as if in the real protocol. For $j \in [1, b], i \in [2^j]$, $\mathcal{S}_{\mathcal{SH}}$ computes the ciphertext $\hat{C}_j$ as an uniformly random string.

We prove the indistinguishability by a series of hybrid games.

- **Game**$_0$: Real protocol (α's sibling roots have good seeds, others bad)
- **Game**$_j$ for $j \in [1, b]$: Replace level-j ciphertexts with random strings

Indistinguishability: $C_i \overset{c}{\approx} \hat{C}_i$ because

$$\begin{aligned}
C_i &= s_i^{\mathcal{G}} \oplus H(v_i^C, Y_{i\oplus 1}^G \oplus \Delta) \\
&\overset{c}{\approx} s_i^{\mathcal{G}} \oplus \mathcal{R}(v_i^C, Y_{i\oplus 1}^G, 1) \quad \text{(CCRH)} \\
&\overset{c}{\approx} \hat{C}_i \quad (\mathcal{R} \text{ is random})
\end{aligned}$$

Also, the derived seeds $\hat{s}_i$ for $i \in [2^n]$ are indistinguishable from the real seeds $s_i^{\mathcal{G}}$ or $s_i^{\mathcal{B}}$ because if i is a sibling root of α (*i.e.*, s_i is a good seed in $\mathbf{Game}_{j-1}$), then

$$\begin{aligned}
\hat{s}_i &= \hat{C}_i \oplus H(v_i^C, Y_{i\oplus 1}^G \oplus \Delta) \\
&\overset{c}{\approx} \hat{C}_i \oplus \mathcal{R}(v_i^C, Y_{i\oplus 1}^G, 1) && \text{Definition 1} \\
&\overset{c}{\approx} (\mathcal{R}(v_i^s, s_{i[0:j-1)}^{\mathcal{G}}, 1) \oplus \mathcal{R}(v_i^C, Y_i^G, 1)) \oplus \mathcal{R}(v_i^C, Y_i^G, 1) && \mathcal{R} \text{ is a random function} \\
&\overset{c}{\approx} H(v_i^s, s_{i[0:j-1)}^{\mathcal{G}}, 1) \\
&= s_i^{\mathcal{G}}
\end{aligned}$$

If i is not a sibling root of α (*i.e.*, s_i is a bad seed in $\mathbf{Game}_{j-1}$), then

$$\begin{aligned}
\hat{s}_i &= \hat{C}_i \oplus H(v_i^C, Y_{i\oplus 1}^G) \\
&\overset{c}{\approx} \hat{C}_i \oplus \mathcal{R}(v_i^C, Y_{i\oplus 1}^G, 0) && \text{Definition 1} \\
&\overset{c}{\approx} (\mathcal{R}(v_i^s, s_{i[0:j-1)}^{\mathcal{G}}, 1) \oplus \mathcal{R}(v_i^C, Y_i^G, 1)) \oplus \mathcal{R}(v_i^C, Y_i^G, 0) && \mathcal{R} \text{ is a random function} \\
&\overset{c}{\approx} (H(v_i^s, s_{i[0:j-1)}^{\mathcal{G}}, 1) \oplus H(v_i^C, Y_i^G, 1)) \oplus H(v_i^C, Y_i^G, 0) \\
&= s_i^{\mathcal{B}}
\end{aligned}$$

Since $s_i^{\mathcal{G}}$ and $s_i^{\mathcal{B}}$ are both indistinguishable from random, they are indistinguishable from each other.
In Conclusion, The view in $\mathbf{Game}_b$ matches the simulator's output $\mathcal{S}_{\mathcal{SH}}$, completing the proof. □

Lemma 3 (Correctness of Table-Only LogStack)**.** TableLogStk *in Figure 11 produces correct outputs.*

Proof. **Goal:** We prove that

$$U^E \oplus U_\alpha^G = \bigoplus_{k\neq\alpha} \pi_{\gamma_k}(T_k)[\beta - \gamma_\alpha] \cdot \Delta$$

Step 1: Decompose U^E. We separate the active branch α from inactive branches:

$$\begin{aligned}
U^E &= \bigoplus_{j\in[1,b]} \bigoplus_{k\in[B]} \langle \hat{T}_{k[0:j)\oplus 1}, Y_k^E \rangle \\
&= \bigoplus_{j\in[1,b]} \langle \hat{T}_{\alpha[0:j)\oplus 1}, Y_\alpha^E \rangle \oplus \bigoplus_{j\in[1,b]} \bigoplus_{k\neq\alpha} \langle \hat{T}_{k[0:j)\oplus 1}, Y_k^E \rangle
\end{aligned}$$

Substituting the values of Y^E, where $Y_\alpha^E = Y_\alpha^G \oplus \mathcal{H}(\beta - \gamma_\alpha) \cdot \Delta$ and $Y_k^E = Y_k^G$ for $k \neq \alpha$:

$$U^E = \bigoplus_{j\in[1,b],k\in[B]} \langle \hat{T}_{k[0:j)\oplus 1}, Y_k^G \rangle \quad \oplus \bigoplus_{j\in[1,b]} \langle \hat{T}_{\alpha[0:j)\oplus 1}, \mathcal{H}(\beta - \gamma_\alpha) \cdot \Delta \rangle$$

Step 2: Analyze the second term. Since $\hat{T}_{\alpha[0:j)\oplus 1}$ uses good seeds (at α's sibling roots):

$$\hat{T}_{\alpha[0:j)\oplus 1} = \hat{T}_{\alpha[0:j)\oplus 1}^{\mathcal{G}} = \bigoplus_{k\neq\alpha} \pi_{\gamma_k}(T_k)$$

Therefore:

$$\bigoplus_{j\in[1,b]} \langle \hat{T}_{\alpha[0:j)\oplus 1}, \mathcal{H}(\beta - \gamma_\alpha) \cdot \Delta \rangle = \bigoplus_{k\neq\alpha} \pi_{\gamma_k}(T_k)[\beta - \gamma_\alpha] \cdot \Delta$$

- PARAMETERS: Parties agree on a 2^n-row m-column public database T and number of branches $B = 2^b$
- INPUT:
  - G inputs $\mathcal{SH}^G$ and Y^G and E inputs $\mathcal{SH}^E$ and Y^E
  - The input satisfies $Y^G \oplus Y^E = [\![\mathcal{H}(\beta - \gamma_\alpha) \otimes \mathcal{H}(\alpha)]\!]$
- OUTPUT:
  - G receives $(U_1^G, \ldots, U_B^G)$ and E receives U^E
  - Constraint: $\forall \alpha, U^E \oplus U_\alpha^G = \bigoplus_{k \neq \alpha} \pi_{\gamma_k}(T_k)[\beta - \gamma_\alpha] \cdot \Delta$
- COMMUNICATION: No communication.
- COMPUTATION: $O((N + Bb)c_\kappa + Nm\kappa + bNm)$ for G and E
- PROCEDURE FOR G:
  - For $j \in [1, b]$ and $i \in [2^j]$, G retrieves $s^{\mathcal{G}}$ and $s^{\mathcal{B}}$ from $\mathcal{SH}^G$ and computes (with $\mathsf{ExpPerm}$ in Figure 9)

$$\hat{T}_i^{\mathcal{G}} \triangleq \mathsf{ExpPerm}(s_i^{\mathcal{G}}) \quad \hat{T}_i^{\mathcal{B}} \triangleq \mathsf{ExpPerm}(s_i^{\mathcal{B}})$$
$$\hat{T}_i^{\mathcal{G}\oplus\mathcal{B}} \triangleq \hat{T}_i^{\mathcal{G}} \oplus \hat{T}_i^{\mathcal{B}}$$

  - G computes $S_{\text{bad}} = \bigoplus_{k \in [B]} \langle \bigoplus_{j \in [1,b]} \hat{T}^{\mathcal{B}}_{k[0:j)}, Y_k^G \rangle$
  - for $j = [1, b]$ and $i \in [2^j]$, G computes

$$\hat{Y}_i^G \triangleq \texttt{if } j = b \texttt{ then } Y_i^G \texttt{ else } \hat{Y}^G_{i||0} \oplus \hat{Y}^G_{i||1}$$

  for $j = [1, b]$ and $i \in [2^j]$, G computes

$$\hat{U}^G \triangleq \begin{cases} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{i\oplus 1}, \hat{Y}_i^G \rangle, & \text{if } j = 1; \\ \hat{U}^G_{i[0:j-1)} \oplus \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{i\oplus 1}, \hat{Y}_i^G \rangle, & \text{otherwise.} \end{cases}$$

  for $i \in [B]$, G outputs $U_i^G = S_{\text{bad}} \oplus \hat{U}_i^G$
- PROCEDURE FOR E:
  - $\forall j \in [1, b], i \in [2^j]$, E gets s_i from $\mathcal{SH}^E$ and computes

$$\hat{T}_i \triangleq \mathsf{ExpPerm}(s_i)$$

  - $\forall j = [1, b], i \in [2^j]$, E computes

$$\hat{Y}_i^E \triangleq \texttt{if } j = b \texttt{ then } Y_i^E \texttt{ else } \hat{Y}^E_{i||0} \oplus \hat{Y}^E_{i||1}$$

  - E outputs $U^E \triangleq \bigoplus_{j \in [1,b], k \in [2^j]} \langle \hat{T}_{k\oplus 1}, \hat{Y}_k^E \rangle$

Figure 11: $\mathsf{TableLogStk}$

Step 3: Analyze the first term. We distinguish good vs. bad tables based on whether they're at α's sibling roots:

$$\begin{aligned} \text{2nd-term} &= \bigoplus_{j \in [1,b], k \in [B]} \langle \hat{T}_{k[0:j)\oplus 1}, Y_k^G \rangle \\ &= \bigoplus_{j \in [1,b], k \in [B]} \begin{cases} \langle \hat{T}^{\mathcal{G}}_{k[0:j)\oplus 1}, Y_k^G \rangle, & k[0{:}j) = \alpha[0{:}j) \\ \langle \hat{T}^{\mathcal{B}}_{k[0:j)\oplus 1}, Y_k^G \rangle, & \text{otherwise} \end{cases} \end{aligned}$$

Recall that $S_{\text{bad}} = \bigoplus_{j \in [1,b], k \in [B]} \langle \hat{T}^{\mathcal{B}}_{k[0:j)\oplus 1}, Y_k^G \rangle$. By XORing S_{bad} to the above term, only the good tables remain. Hence,

$$\begin{aligned}
\text{2nd-term} &= S_{\text{bad}} \oplus \bigoplus_{j\in[1,b],k\in[B]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{\alpha[0:j)\oplus 1}, Y^G_k \cdot \mathbb{1}\{k[1:j) = \alpha[1:j)\}\rangle \\
&= S_{\text{bad}} \oplus \bigoplus_{j\in[1,b]} \langle \hat{T}^{\mathcal{G}\oplus\mathcal{B}}_{\alpha[0:j)\oplus 1}, \hat{Y}^G_\alpha\rangle \\
&= U^G_\alpha
\end{aligned}$$

Conclusion. Combining Steps 1-3: $U^E = U^G_\alpha \oplus \bigoplus_{k\neq\alpha} \pi_{\gamma_k}(T_k)[\beta - \gamma_\alpha] \cdot \Delta$
Therefore: $U^E \oplus U^G_\alpha = \bigoplus_{k\neq\alpha} \pi_{\gamma_k}(T_k)[\beta - \gamma_\alpha] \cdot \Delta$

□

7 Performance

We discuss performance by first stating the analytical costs of our GPIR. We then discuss experimental evaluation and show that it matches the analysis.

In essence, our table-only stacking reduces communication from $O(m(N + \log N \cdot \kappa))$ of `logrow` to $O(m\sqrt{N\kappa})$. For larger N, when the term mN dominates `logrow`'s cost, our improvement is a factor of $O(\sqrt{N/\kappa})$. Concretely, as shown in Figure 12, the break-even point for us is $N = 64$, and e.g., for $N = 2^{25}$ we have >512× smaller communication. Our computation is similar to that of `logrow` for the same problem instance. The total runtime improvement (in terms of wallclock time, as compared to `logrow`) for several network configurations is plotted in Figure 13.

We also compare GPIR to the *fully amortized*, over $O(N)$ accesses, cost of GRAM look-up, which at a minimum is equivalent to 300 AND gates [21]. For a single access, GRAM would incur most of the cost of $O(N)$ accesses, and is not a suitable primitive for the task.

Detailed Performance Analysis Because of the simpler mapping of protocol steps to costs, it is instructive to first present in Theorem 5 the detailed costs of our stacking for a parameterized number of branches (sub-tables). In Corollary 3 we present the costs for the optimal setting of $2^b = \Theta(\sqrt{N/\kappa})$.

Theorem 5. *Our GPIR construction* `Toss` *for a* 2^n*-row* m*-column public database with* 2^b *branches incurs the following costs:*

- $O((bn + (2^b + n)m)\kappa + 2^b(n - b) + 2^{n-b}m)$ *bits of communication.*
- $O((2^n + b2^b)c_\kappa + Nm(b + \kappa))$ *computation.*

Corollary 3. *For a* N*-row* m*-column public database, our PIR gate* `Toss` *achieves the following costs if* $N = o(2^{\kappa - \log\kappa})$*:*

- *Communication Cost:* $O(\sqrt{N}m\sqrt{\kappa})$
- *Computation Cost:* $O(Nm\kappa + (\sqrt{\frac{N}{\kappa}}m + N)c_\kappa)$

7.1 Proofs of Performance Analysis

Proof of Theorem 5. We calculate the cost by analyzing each component of our `Toss` protocol based on the construction in Figure 10. The total cost comprises three main components: SortingHat, demux operations, and the stacked sub-table materials. `Toss` requires G send to E the following number of bits:

$$\begin{aligned}
&(5 \cdot 2^b + (b + 2)n + (2^b + n - b)m - 6)\kappa + 2^b(n - b) + 2^{n-b}m \\
= {}& O((bn + (2^b + n)m)\kappa + 2^b(n - b) + 2^{n-b}m)
\end{aligned}$$

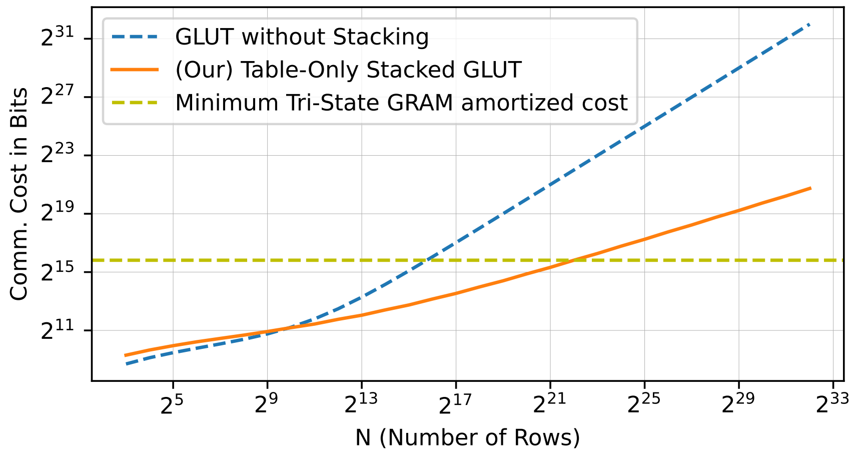

Figure 12: `Toss`'s communication per output bit for $m = 32$-bit items and $\kappa = 128$. `Toss` outperforms `logrow` as soon as $N = 64$ and obtains >512× reduction when $N = 2^{25}$. `Toss` outperforms Tri-State RAM [21] with *amortized* cost when $N \leq 2^{23}$.

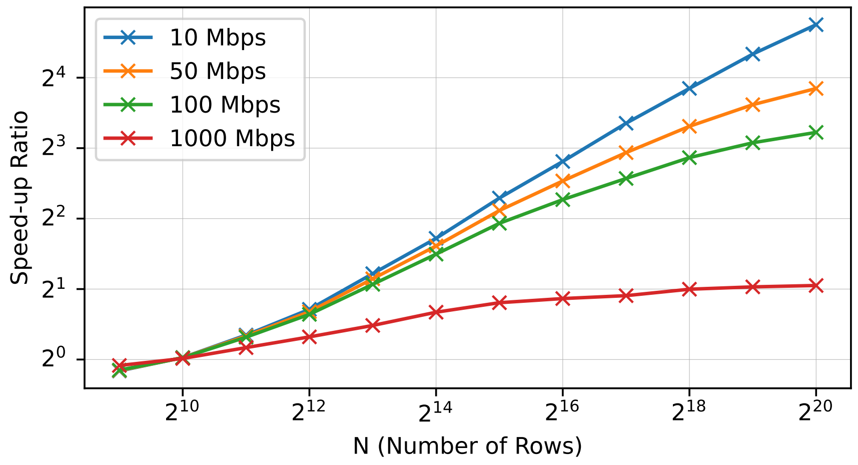

Figure 13: `Toss`'s speed-up ratio on throughput compared to `logrow` under different bandwidth for $m = 16$-bit items and $\kappa = 128$. The speed-up is noticable (2×) since $N = 2^{12}$ and increases as N grows. When $N = 2^{20}$, the speed-up ratio is >32× for a 10Mbps channel and is > 8× for a 100Mbps channel.

Each party has the following computation over bits:

$$O(2^n m(\kappa + b) + 2^b(n + m)c_\kappa + (2^n + b2^b)c_\kappa)$$

$\square$

Proof of Corollary 3. By setting $2^b = \Theta(\sqrt{\frac{N}{\kappa}})$ in Theorem 5, our protocol has

- Communication Cost: $O(\sqrt{N}m\sqrt{\kappa} + \sqrt{N}\log(N\kappa)\frac{1}{\sqrt{\kappa}})$
- Computation Cost: $O(Nm(\kappa + \log\frac{N}{\kappa}) + (\sqrt{\frac{N}{\kappa}}m + N)c_\kappa)$

Then, with the assumption that $N = o(2^{\kappa - \log\kappa})$, we have the desired communication and computation cost. $\square$

Performance Evaluation We implemented GPIR on top of `emp-tool`. As [18] did not implement `logrow`, we implemented it both as a base building block for our protocol and as the evaluation baseline. All experiments (ours and `logrow`) were run on an M3 Pro MacBook with 32 GB memory and deployed 16 threads.

Figure 12 shows the concrete communication cost of GPIR. Figure 13 shows `Toss`'s speed-up ratio compared to `logrow`.

Acknowledgments: This work was partially supported by a Visa research award and NSF awards CNS-2246354 and CCF-2217070.

References

[1] 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

[2] Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 337–367. Springer, Berlin, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_12

[3] Brüggemann, A., Hundt, R., Schneider, T., Suresh, A., Yalame, H.: FLUTE: Fast and secure lookup table evaluations. In: 2023 IEEE Symposium on Security and Privacy. pp. 515–533. IEEE Computer Society Press (May 2023). https://doi.org/10.1109/SP46215.2023.10179345

[4] 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, Berlin, Heidelberg (Mar 2012). https://doi.org/10.1007/978-3-642-28914-9_3

[5] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995. pp. 41–50. IEEE Computer Society (1995)

[6] Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. J. ACM **45**(6), 965–981 (1998)

[7] Doerner, J., shelat, a.: Scaling ORAM for secure computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 523–535. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3133967

[8] Gupta, K., Jawalkar, N., Mukherjee, A., Chandran, N., Gupta, D., Panwar, A., Sharma, R.: SIGMA: secure GPT inference with function secret sharing. Proc. Priv. Enhancing Technol. **2024** (2024)

[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, Cham (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. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part V. LNCS, vol. 14655, pp. 3–31. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58740-5_1

[12] Heath, D., Kolesnikov, V.: A 2.1 KHz zero-knowledge processor with BubbleRAM. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2055–2074. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417283

[13] 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, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56880-1_27

[14] Heath, D., Kolesnikov, V.: Stacked garbling for disjunctive zero-knowledge proofs. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 569–598. Springer, Cham (May 2020). https://doi.org/10.1007/978-3-030-45727-3_19

[15] 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

[16] Heath, D., Kolesnikov, V.: PrORAM - fast $P(\log n)$ authenticated shares ZK ORAM. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part IV. LNCS, vol. 13093, pp. 495–525. Springer, Cham (Dec 2021). https://doi.org/10.1007/978-3-030-92068-5_17

[17] 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, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77883-5_1

[18] Heath, D., Kolesnikov, V., Ng, L.K.L.: Garbled circuit lookup tables with logarithmic number of ciphertexts. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part V. LNCS, vol. 14655, pp. 185–215. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58740-5_7

[19] Heath, D., Kolesnikov, V., Ng, L.K.L.: Garbled circuit lookup tables with logarithmic number of ciphertexts. Cryptology ePrint Archive, Report 2024/369 (2024), `https://eprint.iacr.org/2024/369`

[20] 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, Cham (May / Jun 2022). https://doi.org/10.1007/978-3-031-06944-4_1

[21] Heath, D., Kolesnikov, V., Ostrovsky, R.: Tri-state circuits - A circuit model that captures RAM. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 128–160. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_5

[22] Hou, X., Liu, J., Li, J., Li, Y., jie Lu, W., Hong, C., Ren, K.: CipherGPT: Secure two-party GPT inference. Cryptology ePrint Archive, Report 2023/1147 (2023), `https://eprint.iacr.org/2023/1147`

[23] Kolesnikov, V.: Gate evaluation secret sharing and secure one-round two-party computation. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 136–155. Springer, Berlin, Heidelberg (Dec 2005). https://doi.org/10.1007/11593447_8

[24] 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, Cham (Dec 2018). https://doi.org/10.1007/978-3-030-03332-3_2

[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, Berlin, Heidelberg (Jul 2008). https://doi.org/10.1007/978-3-540-70583-3_40

[26] Kushilevitz, E., Ostrovsky, R.: Replication is NOT needed: SINGLE database, computationally-private information retrieval. In: 38th FOCS. pp. 364–373. IEEE Computer Society Press (Oct 1997). https://doi.org/10.1109/SFCS.1997.646125

[27] Lehmkuhl, R., Mishra, P., Srinivasan, A., Popa, R.A.: Muse: Secure inference resilient to malicious clients. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 2201–2218. USENIX Association (Aug 2021)

[28] 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)

[29] Ng, L.K.L., Chow, S.S.M.: GForce: GPU-friendly oblivious and rapid neural network inference. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 2147–2164. USENIX Association (Aug 2021)

[30] Ng, L.K.L., Chow, S.S.M.: Sok: Cryptographic neural-network computation. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 497–514. IEEE (2023)

[31] Pang, Q., Zhu, J., Möllering, H., Zheng, W., Schneider, T.: BOLT: Privacy-preserving, accurate and efficient inference for transformers. Cryptology ePrint Archive, Report 2023/1893 (2023), `https://eprint.iacr.org/2023/1893`

[32] 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

[33] Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow2: Practical 2-party secure inference. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 325–342. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417274

[34] 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, Cham, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84242-0_5

[35] Wang, X.S., Gordon, S.D., McIntosh, A., Katz, J.: Secure computation of MIPS machine code. In: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K., Meadows, C.A. (eds.) ESORICS 2016, Part II. LNCS, vol. 9879, pp. 99–117. Springer, Cham (Sep 2016). https://doi.org/10.1007/978-3-319-45741-3_6

[36] Yang, Y., Peceny, S., Heath, D., Kolesnikov, V.: Towards generic MPC compilers via variable instruction set architectures (VISAs). In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) ACM CCS 2023. pp. 2516–2530. ACM Press (Nov 2023). https://doi.org/10.1145/3576915.3616664

[37] 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, Berlin, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46803-6_8

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

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

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

Figure 14: Privacy-Free One-Hot Garbling PubOneHot.

Appendix

A Sketch of OuterProduct

We refer readers to [15] for the full formalization. Following is the high-level overview of the construction. Firstly, the parties compute a public one-hot at the masked index, *i.e.*, $[\![\mathcal{H}(\beta - \gamma_\alpha)]\!]$, which we call the master OHG. Then, they obliviously route it to the active branch. The routing is same as how one extends one-hot garbling for outer-product: for each branch, G hashes each zero-share of the master OHG vector to be the 0-label of the local OHG. Similarly, E can obtain the 0-label via hashing except at the masked index. To let E obtains a 1-label at the masked index only if the branch is active, G also sends a encrypted table with two rows: one is the XOR-sum of all these local zero-share, and the other is XOR-sum of these local-share plus Δ. G encrypts these two rows using the private one-hot shares (whose α-th entry is 1) as keys such that E will obtain the XOR-sum with Δ only if the branch is active.

In sum, G only needs to sends $a\kappa$ bits for the public one-hot garbling, and B encrypted tables, each with 2 κ-bit labels. The total costs is $(a + 2B)\kappa$ bits.

B Formal Construction of LogStack

Refining in-mux for LogStack Using private one-hot garbling, we can optimize generic in-mux gadget (i.e. applicable to general stacking), reducing communication overhead from $O(B^2 a\kappa)$ to $O(Ba\kappa)$.

Let $[\![x]\!]$ be the (pre-in-mux) input and $[\![\alpha]\!]$ be the active branch index, where $x \in \{0,1\}^a$, and $\alpha \in \{0,1\}^b$ with $B = 2^b$ branches. Parties first expand $[\![\alpha]\!]$ to a private one-hot garbling $[\![\mathcal{H}(\alpha)]\!]$.

$$\begin{array}{cccccccc} & & & & \alpha\text{-th} & & & \\ & & & & \downarrow & & & \\ \mathcal{H}(\alpha) = & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ & \perp & & \perp & x_1 & \perp & & \perp \\ & & \ddots & & \vdots & & \ddots & \\ & \perp & & \perp & x_a & \perp & & \perp \end{array}$$

Next, they arrange the outputs of the in-mux gadget in an a-by-B matrix as shown above: only the inputs for the active branch (the α-th column) have valid labels; the rest are dummy labels that are always the same regardless of the value of x (but are different for each branch and each input bit). Given $[\![\alpha]\!]$, players compute this table by using $[\![\mathcal{H}(\alpha)]\!]$ and $[\![x]\!]$ as keys to encrypt the desired labels for branches' inputs.

- Parameters: Parties agree on
  - Number of branches $B = 2^b$
  - (Maximum) input size to the sub-circuits a
  - (Maximum) output size from the sub-circuits m
  - Base garbling scheme base for the sub-circuits
  - The sub-circuits $\mathcal{C}_0, \dots, \mathcal{C}_{B-1}$
- Input:
  - Parties inputs a sharing of active branch index $[\![\alpha]\!]$, where $\alpha \in \{0,1\}^b$.
  - Parties inputs a shared input $[\![x]\!]$, where $x \in \{0,1\}^a$.
- Output: Parties receives $[\![\mathcal{C}_\alpha(x)]\!]$, the sharing of the active branch's output.
- Procedure:
  - G and E run $\mathsf{SortingHat}([\![\alpha]\!])$ to obtain $\mathcal{SH}^G$ and $\mathcal{SH}^E$, respectively.
    $\triangleright$ `Commun.:` $(2B-2)\kappa$. `Computation:` $O(Bc_\kappa)$.
  - For $j \in [1, b], i \in [2^j]$, G retrieves $s_i^{\mathcal{G}}$ and $s_i^{\mathcal{B}}$ from $\mathcal{SH}^G$ and computes (with ExpGb in Figure 17)

$$\hat{M}_i^{\mathcal{G}} \triangleq \mathsf{ExpGb}(s_i^{\mathcal{G}}) \quad \hat{M}_i^{\mathcal{B}} \triangleq \mathsf{ExpGb}(s_i^{\mathcal{B}})$$

    $\triangleright$ `Commun.: No. Computation:` $(b+1)B\mathcal{C}'_{\mathsf{comp}} + O(bBc_\kappa)$.
  - G computes and send to E the stacked material $\mathcal{M} = \bigoplus_{i\in[B]} \hat{M}_i^{\mathcal{G}}$
    $\triangleright$ `Commun.:` $\mathcal{C}_{\mathsf{comm}}$. `Comput.:` $O(B\mathcal{C}'_{\mathsf{comm}})$.
  - Parties run $\mathsf{in\text{-}mux}$ to obtains the inputs $\mathbf{Z}^G$ and $\mathbf{Z}^E$, which are the inputs to the sub-garbled-circuits.
    $\triangleright$ `Communication:` $2Ba\kappa$. `Computation:` $O(Bac_\kappa)$.
  - Procedure:

Figure 15: LogStack (continued in Figure 16). This protocol is an straightforward extension of LogStack with their in-mux and SortingHat replaced by our refined protocols. For the simplicity of cost counting, we assume each of the sub-circuit's material is of size $\mathcal{C}'_{\mathsf{comm}}$ and their computation cost for garbling/evaluation cost is $\mathcal{C}'_{\mathsf{comp}}$. The costs are estimated based on the improved in-mux and SortingHat described in this paper.

More specifically, denote by $\mathbf{h}[i]^w$ the label corresponding to wire value $w \in \{0,1\}$ (or w-label) on the i-th entry of $[\![\mathcal{H}(\alpha)]\!]$. Similarly, let $\mathbf{x}[i]^w$ be the w-label of $[\![x[i]]\!]$. Denote by $\mathbf{G}[i][j]$ the i-th input to the j-th branch, and by $\mathbf{G}[i][j]^w$ its w-label, where $w \in \{0,1,\perp\}$. The ciphertexts facilitating computing of for each $\mathbf{G}[i][j]^w$ are

$$\begin{aligned}
C_0 &= \mathbf{G}[i][j]^0 \oplus H(\mathbf{h}[j]^1) \oplus H(\mathbf{x}[i]^0)\\
C_1 &= \mathbf{G}[i][j]^1 \oplus H(\mathbf{h}[j]^1) \oplus H(\mathbf{x}[i]^1)\\
C_2 &= \mathbf{G}[i][j]^\perp \oplus H(\mathbf{h}[j]^0) \oplus H(\mathbf{x}[i]^0)\\
C_3 &= \mathbf{G}[i][j]^\perp \oplus H(\mathbf{h}[j]^0) \oplus H(\mathbf{x}[i]^1)
\end{aligned}$$

Clearly, E can only obtain valid and correct labels for the i-th branch if and only if $[\![\mathcal{H}(\alpha)]\!][i] = 1$, *i.e.*, the branch is active, due to the security of H. Note that the order of the ciphertexts is obfuscated via point-and-permute at the expense of sending 4 (instead of 3) ciphertexts, so the order leaks no information about the wire values. As $\mathbf{G}$'s dimension is $a \times B$, the cost for sending all the ciphertexts is $4Ba\kappa$.

We provide the formal construction in Figure 18.

Construction 2. *LogStack is a garbling scheme (Definition 2) that supports the following gates:*
- *The XOR gates, AND gates, lookup gates, and PIR gates same as the definitions in Construction 1.*
- *Branching gates. It is parameterized over a number of branches B, a base garbling scheme, a (maximum) input size a, a (maximum) output size m, and the sub-circuits $\mathcal{C}_1, \dots, \mathcal{C}_B$. It takes as input the active*

- Procedure (continuous):
  - E unstacked his material by using the seeds from $\mathcal{SH}^E$ and obtains his sub-circuit outputs by enumerating $i \in [B]$:

$$L_{i[1:j)} \triangleq \mathcal{M} \oplus \bigoplus_{j \in [b]} \mathsf{ExpGb}(s_{i[0:j) \oplus 1})$$
$$I_i \triangleq \mathsf{base.Ev}(\mathcal{C}_i, L_{i[0:j)}, \mathbf{Z}^E[i])$$

  and computes the xor-sum of these outputs:

$$U^E = \bigoplus_{i \in [B]} I_i$$

  $\triangleright$ `Comm.: No. Comp.:` $(b+1)B\mathcal{C}'_{\mathsf{comp}} + O(bB\mathcal{C}'_{\mathsf{comm}})$
  - G emulates E's unstacking procedures to predict the possible unstacked materials and the garbage outputs by enumerating $j \in [1, b], i \in [B]$:

$$\hat{L}_{j,i} \triangleq \mathcal{M} \oplus \bigoplus_{k \in [1,j]} \hat{M}^{\mathcal{G}}_{i[0:k) \oplus 1} \oplus \bigoplus_{k \in \{j+1, \ldots, b\}} \hat{M}^{\mathcal{B}}_{i[0:k) \oplus 1}$$
$$\hat{I}_{j,i} \triangleq \mathsf{base.Ev}(\mathcal{C}_i, \hat{L}_{j,i}, \mathbf{Z}^G[i]^{\perp})$$

  compute "shortcut" for sum-xor by enumerating $j \in [1, b], i \in [2^j]$:

$$\hat{U}_{i[0:j)} \triangleq \bigoplus_{k \in [2^{b-j}]} \hat{I}_{j, i[0:j) || k[j:b)}$$

  and finally the xor-summed garbage outputs by enumerating $i \in [B]$:

$$U_i \triangleq \bigoplus_{j \in [b]} \hat{U}_{i[0:j) \oplus 1}$$

  $\triangleright$ `Comm.: No. Comp.:` $bB\mathcal{C}'_{\mathsf{comp}} + O(bB\mathcal{C}'_{\mathsf{comm}})$.
  - Parties now implements $\mathsf{out\text{-}mux}$ to translate the outputs. For each branch index $i \in [B]$, G derives the valid 0- and 1-labels of the i-th sub-circuit as a $m \times 2$ matrix Y_i from $\mathsf{base.ev}(\mathcal{C}_i; s_i^{\mathcal{G}})$ For each output-bit index $j \in [m]$, G sampling a global 0-label $V^G[j]$. Then, for each $i \in [B], j \in [m]$, G sends

$$C^0_{i,j} \triangleq \oplus H(Y_i[j]^0 \oplus U_i[j]) \oplus V^G[j]$$
$$C^1_{i,j} \triangleq \oplus H(Y_i[j]^1 \oplus U_i[j]) \oplus V^G[j] \oplus \Delta$$

  These ciphertexts are point-and-permuted using the color bits of $[\![\alpha]\!]$ and $Y_i[j] \oplus U_i[j]$. E retrieves the ciphertext accordingly using the color bit of $U^E[i]$ and computes for all $j \in [m]$

$$V^E[j] = C^{\mathcal{C}_\alpha(x)[j]} \oplus H(U^E[j])$$

  $\triangleright$ `Comm.:` $2Bm\kappa$. `Comp.:` $O(Bmc_\kappa)$).
  - Parties output $[\![C_\alpha(x)]\!] = \langle\!\langle V^G, V^E \rangle\!\rangle$.

Figure 16: (Continued) LogStack Construction.

branch index $\alpha \in \{0,1\}^{\lceil \log_2 B \rceil}$ *and a value* $x \in \{0,1\}^a$ *and outputs* $\mathcal{C}'_\alpha(x) \in \{0,1\}^m$

- Parameters: The party is given
  - Number of branches $B = 2^b$
  - Base garbling scheme base for the sub-circuits
  - The sub-circuits $\mathcal{C}_1, \dots, \mathcal{C}_B$
- Input: The party input a seed $s_{x[1:j)}$ and implicitly its index $x[1:j)$.
- Output: The parties received a XOR-summed materials $\hat{M}$ derived by expanding $s_{x[1:j)}$. Parties receives $[\![\mathcal{C}_\alpha(x)]\!]$, the sharing of the active branch's output.
- Procedure:
  - The party sets $s_x^{\mathsf{tmp}} = s_x$ and, for $j \in [1, b-k], i \in [2^j])$, computes

$$s_{x||i}^{\mathsf{tmp}} \triangleq H(v_{x||i}^s, s_{x||i[0:j-1)}^{\mathsf{tmp}})$$

  - For any $y \in \{0,1\}^b$ such that $y[0:k-1) = x$, the party computes a garbled material

$$\hat{M}_y^{\mathsf{tmp}} \triangleq \mathsf{base.Gb}(\mathcal{C}'_y; s_y^{\mathsf{tmp}})$$

  - For $j \in [1, b-k], i \in [2^j]$, the party computes $y \triangleq x||i \in \{0,1\}^n$ and

$$\hat{M}_y^{\mathsf{tmp}} \triangleq \hat{M}_{y||0}^{\mathsf{tmp}} \oplus \hat{M}_{y||1}^{\mathsf{tmp}}$$

  - The party receives $\hat{M}_x = \hat{M}_x^{\mathsf{tmp}}$

Figure 17: $\mathsf{ExpGb}(s_x)$: A local procedure that expands the seed, derives the garbled materials, and computes the XOR-sum.

For the garbling algorithms, En and De same as the ones in Construction 1. We define Gb (resp. Ev) same as the one in Construction 1's, but additionally for branch gates, the algorithm runs Figure 15 and Figure 16 with the role of G (resp., E). Gb also appends into $\mathcal{M}$ all messages sent by G to E.

- Parameters: Parties agree on the number of branches $B = 2^b$ and input size n and a base (and stackable) garbling scheme Gb for the sub-circuits.
- Input:
  - G and E input their seeds received from SortingHat, *i.e.*, $\mathcal{SH}^G$ and $\mathcal{SH}^E$, respectively.
  - Parties input a shared (private) one-hot encoding $\langle\langle \mathbf{h}^G, \mathbf{h}^E \rangle\rangle = [\![\mathcal{H}(\alpha)]\!]$ where $\alpha \in \{0,1\}^b$.
  - Parties inputs a shared input $[\![x]\!]$ for translation, where $x \in \{0,1\}^n$.
- Communication: G sends to E $4B\kappa$ bits
- Computation:
  - G uses $O(B(c_\kappa + n \cdot c_{Gb}))$ computation, where c_{Gb} is the cost of generating the garbled labels for an input bit.
  - E uses $O(Bc_\kappa)$ computation
- Output: G and E obtain $n \times B$ array $\mathbf{Z}^G$ and $\mathbf{Z}^E$, respectively. For each input-bit index $i \in [n]$ and each brach index $j \in [B]$,
  - $\mathbf{Z}^G[i][j]$ is a tuple with three values:
    * $Z^0_{i,j}$, a label of wire value 0
    * $Z^1_{i,j}$, a label of wire value 1
    * $Z^\perp_{i,j}$, an invalid label that is a uniform random string
  -

$$\mathbf{Z}^E[i][j] = \begin{cases} Z^{x[j]}_{i,j}, & \text{if } i = \alpha; \\ Z^\perp_{i,j}, & \text{otherwise.} \end{cases}$$

- Procedure:
  - For each branch index $j \in [B]$, G extracts the good seeds $s^{\mathcal{G}}_{j[1:b)}$ from $\mathcal{SH}^G$ and derives $Z^0_{i,j}, Z^1_{i,j}$ according to Gb for all $j \in [n]$.
  - For each branch index $j \in [B]$ and input-bit index $i \in [n]$, G also samples a uniform random string of the same length of $X^0_{i,j}$ (or $X^1_{i,j}$) and assigns it as $\perp_{i,j}$. Then, G sends the following ciphertexts to E

$$\begin{aligned} C^{00}_{i,j} &= Z^\perp_{i,j} \oplus H(v_{i,j,\mathbf{h}}, \mathbf{h}^G[j] \qquad) \oplus H(v_{i,j,\mathbf{x}}, x[i]^G) \\ C^{01}_{i,j} &= Z^\perp_{i,j} \oplus H(v_{i,j,\mathbf{h}}, \mathbf{h}^G[j] \qquad) \oplus H(v_{i,j,\mathbf{x}}, x[i]^G \oplus \Delta) \\ C^{10}_{i,j} &= Z^0_{i,j} \oplus H(v_{i,j,\mathbf{h}}, \mathbf{h}^G[j] \oplus \Delta) \oplus H(v_{i,j,\mathbf{x}}, x[i]^G) \\ C^{11}_{i,j} &= Z^1_{i,j} \oplus H(v_{i,j,\mathbf{h}}, \mathbf{h}^G[j] \oplus \Delta) \oplus H(v_{i,j,\mathbf{x}}, x[i]^G \oplus \Delta) \end{aligned}$$

  The order of these ciphertexts are obfuscated via the point-and-permute technique according to the color bits of $[\![\mathcal{H}(\alpha)[j]]\!]$ and $[\![x[i]]\!]$.
  - For each branch index $j \in [B]$ and input-bit index $i \in [n]$, E retrieves the ciphertext $C^{\mathcal{H}(\alpha)[j]||x[i]}$ via point-and-permute and obtains

$$\begin{aligned} & C^{\mathcal{H}(\alpha)[j]||x[i]} \oplus H(v_{i,j,\mathbf{h}}, \mathbf{h}^E[j]) \oplus H(v_{i,j,\mathbf{x}}, x[i]^E) \\ = & \begin{cases} Z^{x[i]}_{i,j}, & \text{if } \mathcal{H}(\alpha)[i] = 1; \\ Z^\perp_{i,j}, & \text{otherwise.} \end{cases} \end{aligned}$$

Figure 18: Our Lightweight in-mux.