LogStack: Stacked Garbling with $O(b \log b)$ Computation

David Heath and Vladimir Kolesnikov

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

Abstract. Secure two party computation (2PC) of arbitrary programs can be efficiently achieved using garbled circuits (GC). Until recently, it was widely believed that a GC proportional to the entire program, including parts of the program that are entirely discarded due to conditional branching, must be transmitted over a network. Recent work shows that this belief is *false*, and that communication proportional only to the longest program execution path suffices (Heath and Kolesnikov, CRYPTO 20, [HK20a]). Although this recent work reduces needed communication, it *increases* computation. For a conditional with b branches, the players use $O(b^2)$ computation (traditional GC uses only $O(b)$).

Our scheme LogStack reduces stacked garbling computation from $O(b^2)$ to $O(b \log b)$ with *no* increase in communication over [HK20a]. The cause of [HK20a]'s increased computation is the oblivious collection of *garbage labels* that emerge during the evaluation of inactive branches. Garbage is collected by a *multiplexer* that is costly to generate. At a high level, we redesign stacking and garbage collection to avoid quadratic scaling.

Our construction is also more *space efficient*: [HK20a] algorithms require $O(b)$ space, while ours use only $O(\log b)$ space. This space efficiency allows even modest setups to handle large numbers of branches.

[HK20a] assumes a random oracle (RO). We track the source of this need, formalize a simple and natural added assumption on the base garbling scheme, and remove reliance on RO: LogStack is secure in the standard model. Nevertheless, LogStack can be instantiated with typical GC tricks based on non-standard assumptions, such as free XOR and half-gates, and hence can be implemented with high efficiency.

We implemented LogStack (in the RO model, based on half-gates garbling) and report performance. In terms of wall-clock time and for fewer than 16 branches, our performance is comparable to [HK20a]'s; for larger branching factors, our approach clearly outperforms [HK20a]. For example, given 1024 branches, our approach is 31× faster.

Keywords: 2PC, Garbled Circuits, Conditional Branching, Stacked Garbling

1 Introduction

Secure two party computation (2PC) of programs representable as Boolean circuits can be efficiently achieved using garbled circuits (GC). However, circuit-based MPC in general is problematic because conditional control flow does not

have an efficient circuit representation: in the cleartext program, only the taken execution is computed whereas in the circuit *all* branches must be computed.

Until recently, it was assumed that the players must not only compute all branches, but also transmit a string of *material* (i.e., the garbled circuit itself) proportional to the entire circuit. Since communication is the GC bottleneck, transmitting this large string was problematic for programs with conditionals.

Stacked Garbling [HK20a], which we interchangeably call Stacked Garbled Circuit (SGC), shows that expensive branching-based communication is unnecessary: the players need only send enough material for the single longest branch. This single piece of *stacked* material can be re-used across all conditional branches, substantially reducing communication. Unfortunately, this improvement comes with one important downside: SGC requires the players to compute more than they would have without stacking. In particular, for a conditional with b branches, the [HK20a] GC generator must evaluate under encryption each branch $b-1$ times and hence must pay $O(b^2)$ total computation. In contrast, standard garbling uses computation linear in the number of branches.

In this work, we present a new SGC construction that incurs only $O(b \log b)$ computation for both players while retaining the important communication improvement of [HK20a]. The construction also features improved space complexity: while [HK20a] requires the generator to store $O(b)$ intermediate garblings, both `Eval` and `Gen` in our construction use only $O(\log b)$ space. Finally, the construction features low constants and hence opens the door to using SGC even in the presence of high branching factors without prohibitive computation.

1.1 A Case for High Branching Factor

Branching is ubiquitous in programming, and our work significantly improves the secure evaluation of programs with branching. Moreover, the efficient support of *high branching factor* is more important than it may first appear.

Efficient branching enables optimized handling of *arbitrary control flow*, including repeated and/or nested loops. Specifically, we can repeatedly refactor the source program until the program is a single loop whose body conditionally dispatches over straightline fragments of the original program.[1] However, these types of refactorings often lead to conditionals with high branching factor.

As an example, consider a program P consisting of a loop L_1 followed by a loop L_2. Assume the total number of loop iterations T of P is known, as is usual in MPC. For security, we must protect the number of iterations T_1 of L_1 and T_2 of L_2. Implementing such a program with standard Yao GC requires us to execute loop L_1 T times and then to execute L_2 T times. SGC can simply execute $\mathsf{Stack}(L_1, L_2)$ T times, a circuit with a significantly smaller garbling. This observation corresponds to the following refactoring:

$$\texttt{while}(\texttt{e}_0)\{\texttt{s}_0\};\ \texttt{while}(\texttt{e}_1)\{\texttt{s}_1\} \longrightarrow \texttt{while}(\texttt{e}_0 \vee \texttt{e}_1)\{\ \texttt{if}(\texttt{e}_0)\{\texttt{s}_0\}\ \texttt{else}\ \{\texttt{s}_1\}\ \}$$

[1] As a brief argument that this is possible, consider that a CPU has this structure: in this case the 'straightline fragments' are the instruction types handled by the CPU.

where $\mathtt{s_i}$ are nested programs and e_i are predicates on program variables.[2] The right hand side is friendlier to SGC, since it substitutes a loop by a conditional. Now, consider that s_0 and s_1 might themselves have conditionals that can be flattened into a single conditional with all branches. By repeatedly applying such refactorings, even modest programs can have conditionals with high branching factors. High-performance branching, enabled by our approach, allows the efficient and secure evaluation of such programs.

In this work, we do not further explore program refactorings as an optimization. However, we firmly believe that SGC is an essential tool that will enable research into this direction, including CPU emulation-based MPC. As argued above, performance in the presence of high branching factor is essential.

1.2 [HK20a] and its $O(b^2)$ computation

Our approach is similar to that of [HK20a]: we also stack material to decrease communication. The key difference is our reduced computation. It is thus instructive to review [HK20a], focusing on the source of its quadratic scaling.

The key idea of SGC is that the circuit generator `Gen` garbles, starting from seeds, each branch $\mathcal{C}_i$. He then *stacks* these b garbled circuits, yielding only a single piece of material proportional to the longest branch: $M = \bigoplus_i \hat{\mathcal{C}}_i$.[3] Because garblings are expanded from short seeds, the seeds are compact representations of the garblings. Although it would be insecure for the evaluator `Eval` to receive *all* seeds from `Gen`, [HK20a] show that it *is secure* for her to receive seeds corresponding to the inactive branches. Let α be the id of the active branch. `Eval` can reconstruct from seeds the garbling of each inactive branch, use XOR to unstack the material $\hat{\mathcal{C}}_\alpha$, and evaluate $\mathcal{C}_\alpha$ normally. Of course, what is described so far is not secure: the above procedure implies that `Eval` knows α, which she does not in general know and which she should not learn.

Thus, [HK20a] supplies to `Eval` a 'bad' seed for the active branch: i.e., she receives a seed that is different yet indistinguishable from the seed used by `Gen`. From here, `Eval` simply *guesses which branch is taken* (she in fact tries all b branches) and evaluates this guessed branch with the appropriately reconstructed material. For security, each guess is unverifiable by `Eval`. Still, when she guesses right, she indeed evaluates the taken branch and computes valid GC output labels. When she guesses wrong, she evaluates the branch with so-called garbage material (material that is a random-looking string, not an encryption of circuit truth tables), and computes *garbage output labels* (i.e., labels that are not the encryption of 0 or 1, but are random-looking strings). To proceed past the exit of the conditional and continue evaluation, it is necessary to 'collect' these garbage labels by obliviously discarding them in favor of the valid labels.[4]

[2] To be pedantic, this specific refactoring is not always valid: $\mathtt{s_1}$ might mutate variables used in $\mathtt{e_0}$. Still, similar, yet more notationally complex, refactorings are always legal.

[3] Note, [HK20a], as do we in this work, pad each GC material $\hat{\mathcal{C}}_i$ with uniform bits before stacking. This ensures all $\hat{\mathcal{C}}_i$ are of the same length.

[4] Of course, the final output labels of the conditional are fresh, such that they cannot be cross-referenced with those obtained in branch evaluation.

[HK20a] collect garbage without interaction using a garbled gadget called a *multiplexer*. The multiplexer can be non-interactively constructed by `Gen`, but only if he *knows all possible garbage labels*. Once this is satisfied, it is easy for `Gen` to produce a gadget (e.g., appropriate garbled translation tables) that eliminates garbage and propagates the active branch's output labels.

`Gen`'s Uncertainty. It is possible for `Gen` to acquire all garbage labels. [HK20a] achieve this by having `Gen` emulate the actions of `Eval` on all inactive branches. To see how this can be done, consider `Gen`'s knowledge and uncertainty about the garbled evaluation. There are three sources of `Gen`'s uncertainty:

- The input values to each inactive branch. This is the largest source of uncertainty (the number of possibilities are exponential in the number of input wires), but the easiest to handle. [HK20a] introduce a simple trick: they add an additional garbled gadget, the *demultiplexer*, that 'zeros out' the wires into the inactive branches. This fully resolves this source of uncertainty.
- The index of the active branch, which we denote by `truth`.
- `Eval`'s guess of the value of `truth`, which we denote by `guess`.

In total, there are b^2 $(\texttt{truth}, \texttt{guess})$ combinations. Crucially, each of these combinations leads to `Eval` evaluating a *unique combination of a circuit and material*. Hence, there are b^2 possible sets of labels ($b(b-1)$ garbage sets of labels and b valid sets of labels) that the evaluator can compute.

To acquire all possible garbage labels such that he can build the garbage collecting multiplexer, the [HK20a] generator assumes an all-zero inputs for each inactive branch and emulates "in his head" `Eval`'s evaluation of all possible (`truth`,`guess`) combinations. This requires that `Gen` evaluate $b(b-1)$ times on garbage material. This is the source of the $O(b^2)$ computation.

1.3 Top-level Intuition for $O(b \log b)$ Stacked Garbling

Our main contribution is the reduction of SGC computation from $O(b^2)$ to $O(b \log b)$. To this end, we redesign stacking/unstacking to *reduce* `Gen`*'s uncertainty*. By doing so, we reduce the computation needed to implement garbage collection. In this section we provide our highest-level intuition for the construction. Section 2.1 continues in greater detail.

Recall from Section 1.2 the sources of `Gen`'s uncertainty, which result in b^2 evaluations inside `Gen`'s emulation of `Eval`: there are b possible values for both variables `truth` and `guess` ($\texttt{truth} \in \{0, b-1\}, \texttt{guess} \in \{0, b-1\}$). For each fixed pair $(\texttt{truth}, \texttt{guess})$, `Gen` has a fully deterministic view of `Eval`'s garbled evaluation, and hence a deterministic view of the garbage she computes. `Gen` uses the garbage labels to construct the garbage collecting multiplexer.

Our main idea is to consolidate the processing of many such $(\texttt{truth}, \texttt{guess})$ pairs by ensuring that `Eval`'s execution is the same across these $(\texttt{truth}, \texttt{guess})$ pairs. This would further reduce `Gen`'s uncertainty and save computation.

Here is how we approach this. Wlog, let $b = 2^k$ for some $k \in \mathbb{N}$ and consider a balanced binary tree with the b branches at the leaves. For each leaf ℓ, define

the *sibling subtree at level i* (or i-th sibling subtree) to be the subtree rooted in a sibling of the i-th node on the path to ℓ from the tree root. Thus, each branch has $\log b$ sibling subtrees. We call the root of a sibling subtree of a leaf ℓ a *sibling root of* ℓ. Note, the $\log b$ sibling subtrees of a leaf ℓ cover all leaves except for ℓ. For example, consider Figure 1. There, node $\mathcal{C}_3$ has sibling roots $\mathcal{N}_2, \mathcal{N}_{0,1}, \mathcal{N}_{4,7}$.

We reduce the number of possible $(\texttt{truth}, \texttt{guess})$ combinations by changing the semantics of `truth`. `truth` will not denote the active branch. Instead `truth` will now be defined *with respect to a given guess* `guess`. In particular, `truth` will denote the sibling subtree of `guess` that contains the active branch ($\texttt{truth} = 0$ denotes a correct guess). For a fixed `guess`, there are $\log b + 1$ choices for this `truth`. If `Gen` and `Eval` can efficiently process each of these $b \log b$ $(\texttt{truth}, \texttt{guess})$ combinations (they can!), we achieve the improved $O(b \log b)$ computation.

1.4 Our Contributions

[HK20a] shows that GC players need not send a GC proportional to the entire circuit. Instead, communication proportional to only the longest program execution path suffices. However, their improved communication comes at a cost: for a conditional with b branches, the players use $O(b^2)$ computation.

This is a usually a worthwhile trade-off: GC generation is usually much faster than network transmission (cf. our discussion in Section 1.5). However, as the branching factor grows, computation can quickly become the bottleneck due to quadratic scaling. Thus, as we argue in Section 1.1, a more computationally efficient technique opens exciting possibilities for rich classes of problems.

This work presents LogStack, an improvement to SGC that features improved computation without compromising communication. Our contributions include:

- Improved time complexity. For b branches, LogStack reduces time complexity from $O(b^2)$ to $O(b \log b)$.
- Improved space complexity. For b branches, our algorithms require $O(\log b)$ space, an improvement from [HK20a]'s $O(b)$ requirement.
- High concrete performance. In total, the players together garble or evaluate the b branches a total of $\frac{7}{2} b \log b + 2b$ times. These concrete results translate to implementation performance: for fewer than 16 branches, our wall-clock runtime is similar to that of [HK20a]. At higher branching factors, we clearly outperform prior work (see Section 7).
- A formalization in the [BHR12] framework (as modified by [HK20a]) proved secure under standard assumptions. [HK20a] proved SGC secure by assuming a random oracle. We prove security assuming only a pseudorandom function.

1.5 When to use LogStack: a high-level costs consideration

We now informally discuss a broad question of practical importance:

> "If my program has complex control flow, how can I most efficiently implement it for 2PC?"

To make the question more precise, we assume that 'most efficiently' means 'optimized for shortest total wall-clock time'. Since (1) GC is often the most practical approach to 2PC, (2) the GC bottleneck is communication, (3) 'complex control flow' implies conditional behavior, and (4) SGC improves communication for programs with conditional behavior, SGC plays an important role in answering this question. Of course, the cryptographic technique is not the only variable in the optimization space. Program transformations, such as described in Section 1.1, also play a crucial role. These variables are related: some program transformations may lead to a blowup in the number of branches. While SGC alleviates the communication overhead of this blowup, the players still incur $b \log b$ computational overhead. So choosing which program transformations to apply depends also on the performance characteristics of the cryptographic scheme.

Despite the fact that the optimization space for total wall-clock time is complex, we firmly believe the following claim: using LogStack over standard GC will *almost always improve performance.* The rest of this section argues this claim.

Computation vs communication. To discuss how to best apply LogStack, we establish approximate relative costs of GC computation and communication.

Based on our experiments, a commodity laptop running a single core can generate GC material at about 3× the network bandwidth of a 1 Gbps channel. However, while 1Gbps is a typical speed in the LAN setting, WAN speeds are much lower, e.g. 100Mbps. Other network speeds (bluetooth, cellular) are lower still. Even on a LAN and even in a data center, typically we should not assume that our MPC application is allowed to consume the entire channel bandwidth. Rather, we should aim to use as small a fraction of the bandwidth as possible. Based on this discussion, and erring on the conservative side, we choose 100Mbps as "typical" available bandwidth.

Computation is a much more available resource. Today, commodity laptops have four physical cores. Higher-end computing devices, such as desktop CPUs and GPUs have higher numbers of cores and/or per-core processing power, resulting in yet higher GC computation-to-transmission ratio. Precomputation, if available, can also be seen as a way to increase the available compute resource. SGC, even when using our more sophisticated algorithms, is highly parallelizable. It is easy to engage *many* cores to achieve proportional performance improvement. Based on this discussion, and erring on the conservative side, we choose 2 physical cores as a lower end of "typical" available computational power.

Given a typical setting with 2 cores and a 100Mbps channel, we arrive at an approximation that GC computation is $\approx 60\times$ faster than GC transmission.

Assumption: fixed target circuit. To gain a foothold on answering our broad question, we start by ruling out program transformations and consider only cryptographic protocols. Thus, we consider a fixed *baseline circuit* against which we measure SGC and LogStack performance. That is, our baseline is a circuit $\mathcal{C}$ with conditionals, to which we apply garbling scheme directly, and to which we do not apply any program transformations. We may compare 2PC based on LogStack with Yao GC, both instantiated with half-gates [ZRE15].

Rule of thumb: Always apply LogStack. Assuming our approximated speed ratio of GC generation/transmission, and with a few caveats described next, using LogStack for branching will *always* improve over standard GC.

This is easy to see. `Gen` and `Eval` together run a more computationally demanding process, garbling and evaluating branches exactly $\frac{7}{2}b\log b + 2b$ total times ($\frac{5}{2}b\log b + b$ garblings and $b\log b + b$ evaluations). Consider a conditional with b branches. Classic GC will transmit b branches. During this time, `Gen` and `Eval` could have instead performed $60b$ branch garbling/evaluations. LogStack garbles/evaluates $\frac{7}{2}b\log b$ branches. Thus, the point where computation crosses over to become the bottleneck is obtained by solving $\frac{7}{2}b\log b > 60b$, the solution to which is $b \gtrapprox 2^{17} = 131072$ branches. Of course, this is a "rule-of-thumb" estimate and is based on the conservative assumptions discussed above.

If instead a full 1Gbps channel is available (i.e. $10\times$ of our network resource assumption), to arrive at the same cross over point, we would need ten times more cores than our computational resource assumption. That equates to 20 cores; such power is available on mainstream servers.

We conclude that applying LogStack improves wall clock time for nearly all reasonable baseline circuits and settings.

Limits on circuit transformations imposed by computational costs. Above, we established that LogStack is almost always better than standard GC for circuits with branching. It is harder to provide heuristics or even rough suggestions regarding which circuit transformations (cf. in Section 1.1) to apply, and how aggressively they should be applied in conjunction with LogStack secure evaluation. We emphasize that our computational improvement opens a *much* wider optimization space than what was possible with the prior scheme [HK20a]. We leave detailed investigation into this direction as exciting future work.

2 Technical Overview of Our Approach

We now informally present our construction with sufficient detail to introduce the most interesting technical challenges and solutions.

2.1 $O(b \log b)$ Stacked Garbling

Our main contribution is the reduction of SGC computation from $O(b^2)$ to $O(b\log b)$. Our constants are also low: altogether `Gen` issues $\frac{3}{2}b\log b + b$ calls to Gb and b *logb* calls to Ev. `Eval` issues $b\log b$ calls to Gb and b calls to Ev.

We continue the discussion from Section 1.3 in more detail. Our main task is the garbage collection of output labels of *incorrectly guessed* (`truth`, `guess`) combinations where `guess` is `Eval`'s guess of the active branch, and `truth` defines the active branch w.r.t. `guess`. Wlog, let b be a power of 2 to simplify notation. Consider a binary tree where the leaves are the b branches $\mathcal{C}_0, ..., \mathcal{C}_{b-1}$. The tree provides an infrastructure to group branches and to unify processing.

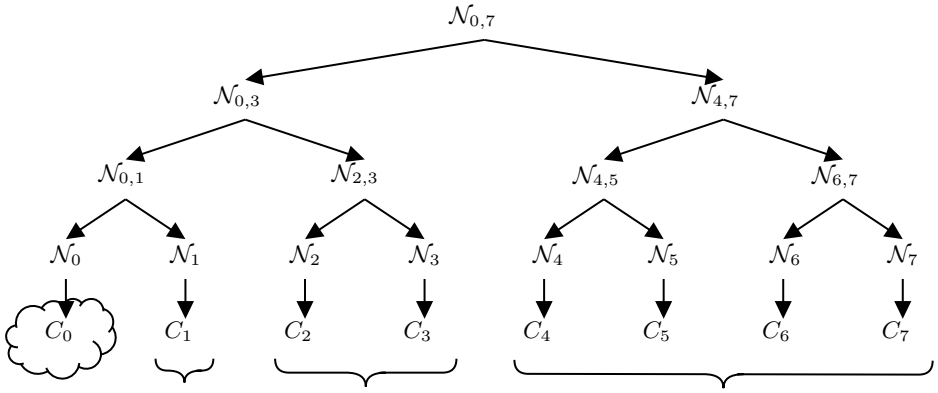

Fig. 1. Suppose there are eight branches $\mathcal{C}_0$ through $\mathcal{C}_7$, and suppose `Eval` guesses that $\mathcal{C}_0$ is the taken branch. If the taken branch is in the subtree $\mathcal{C}_4$ through $\mathcal{C}_7$, `Eval` will generate the same garbage material for the entire subtree, regardless of which branch is actually taken. By extension, $\mathcal{C}_0$ can only be evaluated against $\log 8 = 3$ garbage material strings: one for each sibling subtree (sibling subtrees are bracketed). Hence $\mathcal{C}_0$ has only three possible sets of garbage output labels.

Fix one of b choices for `guess`. In contrast with [HK20a], which then considers b choices for `truth` independently from `guess`, we define `truth` *in relation* to `guess`, and consider fewer `truth` options. Namely, we let `truth` denote the sibling subtree of `guess` that contains the active branch (cf. notation Section 1.3). Given a fixed incorrect `guess`, there are only $\log b$ choices for `truth`.[5] While we have redefined `truth`, the active branch ID α continues to point to the single active branch. Our garbled gadgets compute functions of α.

For concreteness, consider the illustrative example of an 8-leaf tree in Figure 1 where `guess` $= 0$. The discussion and arguments pertaining to this special case generalize to arbitrary b and `guess`.

Consider the four scenarios where one of the branches $\mathcal{C}_4 - \mathcal{C}_7$ is active. These four scenarios each correspond to `truth` $= 1$: $\mathcal{C}_4 - \mathcal{C}_7$ all belong to the level-1 sibling subtree of $\mathcal{C}_0$. We ensure that `Eval`'s unstacking and evaluation in each of these four cases is *identical*, and hence she evaluates the same garbage output labels in these four cases. More generally, we achieve identical processing for all leaves of each sibling subtree. Let α denote the index of the active branch. That is, α is a $\log b$-bit integer that points to the active branch.

Actions and gadgets of `Gen`. In the context of the example in Figure 1, `Gen` garbles branches $\mathcal{C}_0, ..., \mathcal{C}_7$ as follows. Recall, the active branch ID α is available to `Gen` in the form of garbled labels. `Gen` chooses a random seed for the root of the tree (denoted $s_{0,7}$ for the 8-leaf tree in Figure 1), and uses it to pseudorandomly

[5] We focus on garbage collection and consider only incorrect guesses; managing output labels of the correctly guessed branches is straightforward and cheap.

– INPUTS: the active branch id α and the number of branches b.
– OUTPUTS: a sequence of evaluator seeds that form a binary tree:

$$\mathtt{es}_{0,b-1}, \mathtt{es}_{0,\frac{b-1}{2}}, \mathtt{es}_{\frac{b-1}{2}+1,b-1}, \ldots, \mathtt{es}_0, \mathtt{es}_1, \ldots \mathtt{es}_{b-1}$$

such that for each node $\mathcal{N}$:

$$\mathtt{es}_{\mathcal{N}} = \begin{cases} s_{\mathcal{N}}, & \text{if } \mathcal{N} \text{ is a sibling root of } \alpha \\ s'_{\mathcal{N}}, & \text{otherwise} \end{cases}$$

where $s'_{\mathcal{N}}$ is a uniform string indistinguishable from $s_{\mathcal{N}}$.

Fig. 2. The `SortingHat` functionality. `SortingHat` is responsible for conveying only the sibling root seeds of α to `Eval`. For every other node, `Eval` obtains a different, but indistinguishable, seed that, when garbled, generates garbage material. `SortingHat` is easily implemented as a garbled circuit gadget (i.e., built from garbled rows).

derive seeds for each node of the tree. This is done in the standard manner, e.g., the immediate children of a seed s are the PRF evaluations on inputs 0 and 1 with the key s. `Gen` uses each leaf seed s_i to garble the corresponding branch $\mathcal{C}_i$ and stacks all garbled branches $M = \bigoplus_i \hat{\mathcal{C}}_i$. This material M is the large string that `Gen` ultimately sends across the network to `Eval`. We note two facts about M and about the active branch α.

1. **Correctness**: if `Eval` obtains the $\log b$ seeds of the sibling roots of α, then she can regarble all circuits $\hat{\mathcal{C}}_{i \neq \alpha}$, unstack by XORing with M, and obtain $\hat{\mathcal{C}}_\alpha$, allowing her to correctly evaluate $\mathcal{C}_\alpha$.
2. **Security**: `Eval` must not obtain any correct seed corresponding to any ancestor of α. If she did, she would learn (by garbling) the encoding of wire labels which would allow her to decrypt all intermediate wire values in $\mathcal{C}_\alpha$. Instead, `Eval` will obtain 'garbage' seeds indistinguishable yet distinct from the correct seeds generated by `Gen`.

To facilitate garbled evaluation of the conditional and meet the requirements of these two facts, in addition to M, `Gen` generates and sends to `Eval` a small (linear in the number of branches with small constants) garbled gadget that we call `SortingHat`.[6] `SortingHat` aids `Eval` in her reconstruction of branch material. `SortingHat` takes as input labels corresponding to α and produces candidate seeds for each node in the tree. For each node $\mathcal{N}$, `SortingHat` constructs a correct seed $s_{\mathcal{N}}$ if and only if $\mathcal{N}$ is a sibling root of the leaf α (see Figure 2). `SortingHat` can be implemented as a collection of garbled rows. Importantly, since this is a fixed gadget, when evaluated on a node $\mathcal{N}$ that is not a sibling root of α, `Eval` will obtain a *fixed* seed that is predictable to `Gen`.

[6] In J.K. Rowling's Harry Potter universe, the 'sorting hat' is a magical object that assigns new students to different school houses based on personality. Our `SortingHat` 'sorts' nodes of trees into two categories based on α: those that are 'good' (i.e., sibling roots of α) and those that are 'bad'.

For example in Figure 1, if the active branch is $\alpha = 4$, then applying `SortingHat` to nodes $\mathcal{N}_{0,3}, \mathcal{N}_{6,7}, \mathcal{N}_5$ reconstructs the correct seeds $s_{0,3}, s_{6,7}, s_5$. Applying `SortingHat` to other nodes constructs fixed garbage seeds. If instead $\alpha = 3$, then `SortingHat` reconstructs the correct seeds $s_{4,7}, s_{0,1}, s_2$. Critically, the garbage seeds reconstructed in both cases, e.g. for node $\mathcal{N}_{4,5}$, are the same.

Actions of `Eval`. It is now intuitive how `Eval` proceed with unstacking. She applies `SortingHat` and obtains a tree of random-looking seeds; of $2b$ seeds, only $\log b$ seeds just off the path to α (corresponding to α's sibling roots) are correct. `Eval` guesses `guess`; assuming `guess`, she uses only the sibling seeds of `guess` to derive all $b-1$ leaf seeds not equal to `guess`. She then garbles the $b-1$ branches $\mathcal{C}_i$ and unstacks the corresponding GCs $\hat{\mathcal{C}}_i$.

If $\texttt{guess} = \alpha$, `Eval` derives the intended leaf seeds $s_{i\neq\alpha}$, unstacks the intended garbled circuits $\hat{\mathcal{C}}_{i\neq\alpha}$, and obtains the correct GC $\hat{\mathcal{C}}_\alpha$. Consider the case where `Eval` guesses wrong. `Eval` simply unstacks wrong branches garbled with the wrong seeds. Since `Eval` never receives any additional valid seeds, there is no security loss. We next see that the number of different garbage labels we must collect is small, and further that they can be collected efficiently.

***O*(*b* log *b*) computational cost accounting.** Let `Gb` and `Ev` be procedures that respectively garble/evaluate a GC. Consider how many such calls are made by `Eval`. Consider branch $\mathcal{C}_i$. It is garbled $\log b$ times, once with a seed (ultimately) derived from each seed on the path to the root. Thus, the total number of calls by `Eval` to `Gb` is $b \log b$ and to `Ev` is exactly b.

To construct the garbage collecting multiplexer, `Gen` must obtain all possible garbage labels. We demonstrate that the total cost to the generator is $O(b \log b)$ calls to both `Gb` and `Ev`. First, consider only `Gb` and consider the number of ways `Eval` can garble a specific circuit $\mathcal{C}_i$. Clearly, this is exactly $\log b + 1$.

Now, consider `Gen`'s number of calls to `Ev`. Recall that our goal was to ensure that `Eval` constructs the same garbage output labels for a branch $\mathcal{C}_i$ in each scenario where α is in some fixed sibling subtree of $\mathcal{C}_i$. The logic of `SortingHat` ensures that `Eval` obtains the same sibling root seeds in each of these scenarios, and therefore she constructs the same garblings. Hence, since there are $\log b$ sibling subtrees of $\mathcal{C}_i$, $\mathcal{C}_i$ has only $\log b$ possible garbage output labels. Thus, in order to emulate `Eval` in all settings and obtain all possible garbage output labels, `Gen` must garble and evaluate each branch $\log b$ times.

2.2 Technical difference between our and [HK20a] binary braching

A careful reader familiar with [HK20a] may notice that they present two versions of stacked garbling. The first handles high branching factors by recursively *nesting* conditionals. Nested conditionals can be viewed as a binary tree. This first approach is then discarded in favor of a second, more efficient vector approach. Our work advocates binary branching and yet substantially improves over [HK20a]'s vectorized approach. Why is our binary branching better?

The problem with [HK20a]'s recursive construction is that `Eval` recursively garbles the garbage-collecting multiplexer for nested sub-conditionals. Doing so leads to a recursive emulation whereby `Eval` emulates herself (and hence `Gen` emulates himself as well). This recursion leads to quadratic cost for both players. The way out is to treat the multiplexer separately, and to opt not to stack it. If multiplexers are not stacked, then `Eval` need not garble them, and hence `Eval` need never emulate herself. On top of this, we reduce the number of ways that individual branches can be garbled via our `SortingHat`.

A note on nested branches. Nested branches with complex sequencing of instructions emerge naturally in many programs. Our approach operates directly over vectors of circuits and treats them as binary trees. This may at first seem like a disadvantage, since at the time the first nested branching decision is made, it may not yet be possible to make *all* branching decisions. There are two natural ways LogStack can be used in such contexts:

1. Although we advocate for vectorized branching, LogStack does support nested evaluation. Although nesting is secure and correct, we do not necessarily recommend it. Using LogStack in this recursive manner yields quadratic computation overhead.
2. Refactorings can be applied to ensure branches are vectorized. For example, consider the following refactoring:

$$\begin{aligned}&\texttt{if } (\mathtt{e}_0)\ \{\ \mathtt{s}_0; \texttt{if } (\mathtt{e}_1)\ \{\ \mathtt{s}_1\ \}\ \texttt{else}\ \{\ \mathtt{s}_2\ \}\ \}\ \texttt{else}\ \{\ \mathtt{s}_3; \mathtt{s}_4\ \} \longrightarrow \\ &\quad \texttt{if } (\mathtt{e}_0)\ \{\ \mathtt{s}_0\ \}\ \texttt{else}\ \{\ \mathtt{s}_3\ \}; \texttt{switch}(\mathtt{e}_0 + \mathtt{e}_0\mathtt{e}_1)\ \{\ \mathtt{s}_4\ \}\ \mid\ \{\ \mathtt{s}_2\ \}\ \mid\ \{\ \mathtt{s}_1\ \}\end{aligned}$$

   Where s_i are programs, e_i are predicates on program variables, and where s_0, s_3 do not modify variables in e_0. This refactoring has replaced a nested conditional by a sequence of two 'vectorized' conditionals, and hence made the approach amenable to our efficient algorithms.

2.3 Memory Efficiency of LogStack

The [HK20a] approach forces `Gen` to store many intermediate garblings: for conditionals with b branches he requires $O(b)$ space. In contrast, LogStack has low space requirements: its algorithms run in $O(\log b)$ space. We briefly discuss why [HK20a] requires linear space and how our approach improves this.

In the [HK20a] approach, `Eval` obtains $b-1$ good seeds for all but the active branch and a bad seed for the active branch. When `Eval` then makes a particular `guess`, she attempts to uncover the material for `guess` by XORing the stacked material (sent by `Gen`) with $b-1$ reconstructed materials; she 'unstacks' her $b-1$ materials corresponding to all branches that are not equal to `guess`. Recall that `Gen` emulates `Eval` for all combinations of $(\texttt{truth}, \texttt{guess})$ where $\texttt{truth} \neq \texttt{guess}$ to compute garbage outputs. The most intuitive way to proceed, and the strategy [HK20a] uses, is for `Gen` to once and for all garble all circuits using the 'good' seeds and garble all circuits using the 'bad' seeds, and to store all materials in

two large vectors. Let M_i be the good material for a branch $\mathcal{C}_i$ and let M'_i be the bad material. Now let $j = \texttt{truth}$ and $k = \texttt{guess}$. To emulate all possible bad evaluations, $\texttt{Gen}$ evaluates $\mathcal{C}_k$ using the material $M_k \oplus M_j \oplus M'_j$: i.e., he emulates $\texttt{Eval}$ when correctly unstacking all material except M_k (which she will not attempt to unstack because she wishes to evaluate $\mathcal{C}_k$) and M_j (which she attempts to unstack, but fails and instead adds M'_j). Because $\texttt{Gen}$ considers all j, k combinations, it is not clear how $\texttt{Gen}$ can compute all values $M_k \oplus M_j \oplus M'_j$ without either (1) storing intermediate garblings in $O(b)$ space or (2) repeatedly garbling each branch at great cost. [HK20a] opts for the former.

In contrast, because of $\mathsf{LogStack}$'s binary tree structure, we can eagerly stack material together as it is constructed to save space. E.g., consider again the example in Figure 4 where $\texttt{Eval}$ guesses that $\mathcal{C}_0$ is active. Recall, she garbles the entire right subtree starting from the seed for node $\mathcal{N}_{4,7}$, and $\texttt{Gen}$ emulates this same behavior with the bad seed. For both players, the material corresponding to individual circuits, say M_4 corresponding to $\mathcal{C}_4$, is *not interesting or useful*. Only the stacked material $M_4 \oplus .. \oplus M_7$ is useful for guessing $\mathcal{C}_0$ (and more generally for guessing all circuits in the subtree $\mathcal{N}_{0,3}$). Thus, instead of storing all material separately, the players both XOR material for subtrees together as soon as it is available. This trick is the basis for our low space requirement.

There is one caveat to this trick: the 'good' garbling of each branch $\mathcal{C}_i$ *is* useful throughout $\texttt{Gen}$'s emulation of $\texttt{Eval}$. Hence, the straightforward procedure would be for $\texttt{Gen}$ to once and for all compute the good garblings of each branch and store them in a vector, consuming $O(b)$ space. This is viable, and indeed has lower runtime constants than presented elsewhere in this work: $\texttt{Gen}$ would invoke Gb only $b \log b + b$ times. We instead trade in some concrete time complexity in favor of dramatically improved space complexity. $\texttt{Gen}$ garbles the branches using good seeds an extra $\frac{1}{2} b \log b$ times, and hence calls Gb a total of $\frac{3}{2} b \log b + b$ times. These extra calls to Gb allow $\texttt{Gen}$ to avoid storing a large vector of materials, and our algorithms run in $O(\log b)$ space.

2.4 Stacked Garbling with and without Random Oracles

[HK20a] (and we) focus only on branching and leave the handling of low level gates to another *underlying* garbling scheme, Base. [HK20a] assumes nothing about Base except that it satisfies the standard [BHR12] properties, as well as their *stackability* property. However, they do not preclude Base's labels from being related to each other, which presents a security problem: Base's labels are used to garble rows, but if the labels are related they cannot be securely used as PRF keys. [HK20a] handles the possible use of related keys by using a RO.

We introduce a stronger requirement on Base, which we call *strong stackability*. Informally, we additionally require that *all* output labels of Base are uniformly random. This is sufficient to prove security in the standard model.

Of course, RO-based security theorems and proofs also work, and our gadgets could be slightly optimized in a natural manner under this assumption.

3 Related Work

GC is the most popular and often the fastest approach to secure two-party computation. Until recently, it was believed that it is necessary to transmit the entire GC during 2PC, even for inactive conditional branches. Recent breakthrough work [HK20a] showed that this folklore belief is false, and that it suffices to only transmit GC material proportional to the longest execution path.

We focus our comparison with prior work on [HK20a], and then review other related work, such as universal circuits and earlier stacked garbling work.

Comparison with [HK20a]. As discussed in Section 1.1, programs with conditionals with high branching factor may be a result of program transformations aimed at optimizing GC/SGC performance. While the protocol of [HK20a] is concretely efficient, its quadratic computational cost presents a limitation even in settings with relatively modest branching factor b. This significantly limits the scope of program transformations which will be effective for SGC.

Our work achives total computational cost proportional to $3.5b \log b$, and effectively removes the computational overhead of the SGC technique as a constraining consideration[7], as discussed in Section 1.5.

Memory management is a significant performance factor in GC in general, and in particular in [HK20a] garbling. Retrieving an already-garbled material from RAM may take similar or longer time than regarbling from scratch while operating in cache. In addition to significantly improving computation (i.e. number of calls to Gb and Ev), our approach offers improved memory utilization (see Sections 1.4 and 2.3). [HK20a] requires that a linear number of garbled circuits be kept in RAM. For larger circuits this can become a problem. For example, the garbling of a 1M AND-gate circuit occupies 32MB in RAM. If a machine can dedicate 2GB to garbling, a maximum of 64 branches of this size can be handled. This ignores additional constant space costs, which are not necessarily low. In contrast, we use only $O(\log b)$ space, and hence can fit the garblings of large numbers of branches into memory. In our experiments, we ran our implementation on a circuit with 8192 SHA-256 branches, a circuit that altogether holds $>$ 385M AND-gates. Our peak memory usage was at around 100MB ([HK20a] would require more than 12GB of space to run this experiment).

In sum, as discussed at length in Sections 1.5, 2.3 and 7, we essentially eliminate the concern of increased computation due to Stacked Garbling for typical settings and open the door to the possibility of applying a large class of efficiency-improving program transformations on the evaluated program.

Universal circuits. An alternate technique for handling conditional branching is to implement a *universal circuit* [Val76], which can represent any conditional branch. We discuss universal circuits [LMS16,KS16,GKS17,ZYZL19,AGKS20,KKW17]

[7] We stress that branches must still be garbled, and extreme program transformations, such as stacking *all* possible program control flows, may be impractical computationally due to the exponential number of branches.

in more detail in the full version of this paper. In short, SGC is a more practical approach to conditional branching in most scenarios.

Other related work. Kolesnikov [Kol18] was the first to separate the GC *material* from circuit topology. This separation was used to improve GC branching given that the GC generator `Gen` knows the active branch. Subsequently, [HK20b] considered a complementary setting where the GC evaluator `Eval` knows the active branch, and used it to construct efficient ZK proofs for circuits with branching. Our work follows the line of work initiated by [Kol18,HK20b]; it is for general 2PC and is constant-round.

As discussed in [HK20a], interaction, such as via the output selection protocol of [Kol18], can be used to collect garbage efficiently (computation linear in b). However, a communication round is added for each conditional branch. In many scenarios, non-interactive 2PC (such as what we achieve) is preferred.

Designing efficient garbling schemes under standard assumptions (i.e. using only PRFs) is a valuable research direction. [GLNP15] impressively implement garbled table generation and evaluation with speed similar to that of fixed-key AES. [GLNP15] cannot use the Free XOR technique [KS08], which requires circularity assumptions [CKKZ12], but nevertheless implement XOR Gates with only one garbled row and AND gates with two rows.

4 Notation and Assumptions

Notation. Our notation is mostly consistent with the notation of [HK20a].

- Our garbling scheme is called LogStack. We sometimes refer to it by the abbreviation LS, especially when referring to its algorithms.
- '`Gen`' is the circuit generator. We refer to `Gen` as he, him, his, etc.
- '`Eval`' is the circuit evaluator. We refer to `Eval` as she, her, hers, etc.
- '$\mathcal{C}$' is a circuit. $\texttt{inpSize}(\mathcal{C})$ and $\texttt{outSize}(\mathcal{C})$ respectively compute the number of input/output wires to $\mathcal{C}$.
- $x \mid y$ denotes the concatenation of strings x and y.
- Following SGC terminology introduced by [Kol18], M refers to GC *material*. Informally, material is just a collection of garbled tables, i.e. the garbling data which, in conjunction with circuit topology and input labels, is used to compute output labels.
- We use m to denote the size of material, i.e. $m = |M|$.
- Variables that represent vectors are denoted in bold, e.g. $\boldsymbol{x}$. We index vectors using bracket notation: $\boldsymbol{x}[0]$ accesses the 0th index of $\boldsymbol{x}$.
- We extensively use binary trees. Suppose t is such a tree. We use subscript notation t_i to denote the ith leaf of t. We use pairs of indexes to denote internal nodes of the tree. I.e., $t_{i,j}$ is the root of the subtree containing the leaves $t_i..t_j$. $t_{i,i}$ (i.e. the node containing only i) and t_i both refer to the leaf: $t_{i,i} = t_i$. It is sometimes convenient to refer to a (sub)tree index abstractly. For this, we write $\mathcal{N}_{i,j}$ or, when clear from context, simply write $\mathcal{N}$.

– We write $a \leftarrow_{\$} S$ to denote that a is drawn uniformly from the set S.
– $\stackrel{c}{=}$ denotes computational indistinguishability.
– κ denotes the computational security parameter and can be understood as the length of PRF keys (e.g. 128).

We evaluate GCs with input labels that are generated independently of the GC material and do not match the GC. We call such labels *garbage labels*. During GC evaluation, garbage labels propagate to the output wires and must eventually be obliviously dropped in favor of valid labels. We call the process of canceling out output garbage labels *garbage collection*.

Assumptions. LogStack is secure in the standard model. However, higher efficiency of both the underlying scheme Base and of our garbled gadgets can be achieved under the RO assumption. Our implementation uses half-gates as Base, and relies on a random oracle (RO).

5 The LogStack Garbling Scheme

In this section, we formalize our construction, LogStack. Throughout this section, consider a conditional circuit with b branches. For simplicity, we ignore the number input and output wires.

We adopt the above simplification because branching factor is the most interesting aspect of LogStack. We emphasize that ignoring inputs/outputs does not hide high costs. While we scale with the product of the number of inputs and b (and respectively the product of number of outputs and b), the constants are low (see Section 7 for evidence). Thus, inputs/outputs are of secondary concern to the circuit size, which is often far larger than the number of inputs/outputs.

Consider garbled circuits $\hat{\mathcal{C}}_i$ corresponding to each branch $\mathcal{C}_i$. Let m be the size of the largest such garbling: $m = \max_i |\hat{\mathcal{C}}_i|$. Given branching factor b, LogStack features:

– $O(m)$ communication complexity.
– $O(mb \log b)$ time complexity.
– $O(m \log b)$ space complexity.

LogStack is formalized as a *garbling scheme* [BHR12]. Garbling schemes abstract the details of GC such that protocols can be written generically. That is, LogStack is a modular collection of algorithms, not a protocol. Our formalization specifically uses the modified garbling scheme framework of [HK20a], which separates the *topology* of circuits (i.e., the concrete circuit description) from circuit material (i.e., the collections of encryptions needed to securely evaluate the circuit), an important modification for SGC.

A garbling scheme is a tuple of five algorithms:

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

– ev specifies circuit semantics. For typical approaches that consider only low-level gates, ev is often left implicit since its implementation is generally understood. We explicate ev to formalize conventions of conditional evaluation.
– Ev specifies how `Eval` securely evaluates the GC.
– Gb specifies how `Gen` garbles the GC.
– En and De specify the translation of cleartext values to/from GC labels. That is, En specifies how player inputs translate to input labels and De specifies how outputs labels translate to cleartext outputs.

Correct garbling schemes ensure that the garbled functions Gb, En, Ev, and De achieve the semantics specified by ev.

Before we present our garbling scheme LogStack, we introduce the formal syntax of the circuits it manipulates. Because our focus is conditional branching, we assume an *underlying garbling scheme* Base. Base is responsible for handling the collections of low level gates (typically AND and XOR gates) that we refer to as *netlists*. In our implementation, we instantiate Base with the efficient half-gates scheme of [ZRE15]. We do not specify the syntax of netlists, and entirely leave their handling to Base. Our circuit syntax is defined inductively: Let $\mathcal{C}_0, \mathcal{C}_1$ be two arbitrary circuits and $\boldsymbol{\mathcal{C}}$ be a a vector of arbitrary circuits. The space of circuits is defined as follows:

$$\mathcal{C} ::= \mathsf{Netlist}(\cdot) \mid \mathsf{Cond}(\boldsymbol{\mathcal{C}}) \mid \mathsf{Seq}(\mathcal{C}_0, \mathcal{C}_1)$$

That is, a circuit is either (1) a netlist, (2) a conditional dispatch over a vector of circuits (our focus), or (3) a sequence of two circuits. Sequences of circuits are necessary to allow arbitrary control flow.

With our syntax established, we are ready to present our algorithms.

Construction 1 (LogStack)**.** *LogStack is the tuple of algorithms:*

$$(\mathit{LS.ev}, \mathit{LS.Ev}, \mathit{LS.Gb}, \mathit{LS.En}, \mathit{LS.De})$$

Definitions for each algorithm are listed in Figure 3.

We discuss correctness and security of Construction 1 in Section 6. Due to lack of space, proofs of these properties are in the full version of this paper.

In terms of efficiency, LogStack satisfies the following property:

Theorem 1. *Let Base be a garbling scheme satisfying the following property:*

– *Let $\mathcal{C}$ be an arbitrary netlist and let s be the size of material generated by invoking Base.Gb on $\mathcal{C}$. Let both Base.Ev and Base.Gb, invoked on $\mathcal{C}$, run in $O(s)$ time and $O(s)$ space.*

Then Construction 1 instantiated with Base satisfies the following property.

– *Let $\boldsymbol{\mathcal{C}}$ be a vector of b arbitrary netlists. Let m be the maximum size of the garblings constructed by calling Base.Gb on each of these b netlists. Then both LS.Ev and LS.Gb, invoked on Cond($\boldsymbol{\mathcal{C}}$), run in $O(mb \log b)$ time and $O(m \log b)$ space.*

$\mathsf{LS.ev}(\mathcal{C}, \boldsymbol{x})$:
▷ What are the circuit semantics?
`switch` $\mathcal{C}$:
`case` $\mathsf{Netlist}(\cdot)$: `return` $\mathsf{Base.ev}(\mathcal{C}, \boldsymbol{x})$
`case` $\mathsf{Seq}(\mathcal{C}_0, \mathcal{C}_1)$: `return` $\mathsf{LS.ev}(\mathcal{C}_1, \mathsf{LS.ev}(\mathcal{C}_0, \boldsymbol{x}))$
`case` $\mathsf{Cond}(\boldsymbol{\mathcal{C}})$:
▷ split branch index from input
$\alpha \mid \boldsymbol{x}' \leftarrow \boldsymbol{x}$
▷ Run the active branch.
`return` $\mathsf{LS.ev}(\boldsymbol{\mathcal{C}}[\alpha], \boldsymbol{x}')$

$\mathsf{LS.Ev}(\mathcal{C}, M, \boldsymbol{X})$:
▷ How does `Eval` evaluate the GC?
`switch`$(\mathcal{C})$:
`case` $\mathsf{Netlist}(\cdot)$: `return` $\mathsf{Base.Ev}(\mathcal{C}, M, \boldsymbol{X})$
`case` $\mathsf{Seq}(\mathcal{C}_0, \mathcal{C}_1)$:
$M_0 \mid M_{tr} \mid M_1 \leftarrow M$
`return` $\mathsf{LS.Ev}(\mathcal{C}_1, M_1, \mathit{trans}.\mathsf{Ev}(\mathsf{LS.Ev}(\mathcal{C}_0, M_0, \boldsymbol{X}), M_{tr})$
`case` $\mathsf{Cond}(\boldsymbol{\mathcal{C}})$: `return` $\mathtt{EvCond}(\boldsymbol{\mathcal{C}}, M, \boldsymbol{X})$

$\mathsf{LS.En}(e, \boldsymbol{x})$:
▷ How do inputs map to labels?
▷ This works for all projective schemes:
$\boldsymbol{X} \leftarrow \lambda$
`for` $i \in 0..\mathtt{inpSize}(\mathcal{C})-1$:
$(X^0, X^1) \leftarrow e[i]$
`if` $\boldsymbol{x}[i] = 0$: $\{\ \boldsymbol{X}[i] \leftarrow X^0\ \}$ `else` : $\{\ \boldsymbol{X}[i] \leftarrow X^1\ \}$
`return` $\boldsymbol{X}$

$\mathsf{LS.Gb}(1^\kappa, \mathcal{C}, S)$
▷ How does `Gen` garble the GC?
▷ S is an explicit seed.
`switch` $\mathcal{C}$:
`case` $\mathsf{Netlist}(\cdot)$:
`return` $\mathsf{Base.Gb}(1^\kappa, \mathcal{C}, S)$
`case` $\mathsf{Seq}(\mathcal{C}_0, \mathcal{C}_1)$:
▷ Derive seeds for two circuits.
$S_0 \leftarrow F_S(0)$
$S_1 \leftarrow F_S(1)$
$(M_0, e_0, d_0) \leftarrow \mathsf{LS.Gb}(1^\kappa, \mathcal{C}_0, S_0)$
$(M_1, e_1, d_1) \leftarrow \mathsf{LS.Gb}(1^\kappa, \mathcal{C}_1, S_1)$
▷ Labels out of $\mathcal{C}_0$ must be *translated*
▷ to labels into $\mathcal{C}_1$.
$M_{tr} \leftarrow \mathit{trans}.\mathsf{Gb}(d_0, e_1)$
$M \leftarrow M_0 \mid M_{tr} \mid M_1$
`return` (M, e_0, d_1)
`case` $\mathsf{Cond}(\boldsymbol{\mathcal{C}})$: `return` $\mathtt{GbCond}(\boldsymbol{\mathcal{C}}, S)$

$\mathsf{LS.De}(d, \boldsymbol{Y})$:
▷ How do labels map to outputs?
▷ This works for all projective schemes:
$\boldsymbol{y} \leftarrow \lambda$
`for` $i \in 0..\mathtt{outSize}(\mathcal{C})-1$:
$(Y^0, Y^1) \leftarrow d[i]$
`if` $\boldsymbol{Y}[i] = Y^0$: $\boldsymbol{y}[i] \leftarrow 0$
`else if` $\boldsymbol{Y}[i] = Y^1$: $\boldsymbol{y}[i] \leftarrow 1$
`else` : `ABORT`
`return` $\boldsymbol{y}$

Fig. 3. Our garbling scheme LogStack. The included algorithms are typical except for the handling of conditionals. Ev and Gb delegate the core of our approach: `EvCond` (Figure 5) and `GbCond` (Figure 6).

Standard garbling schemes, e.g. the half-gates scheme [ZRE15], achieve the efficiency required by Theorem 1, since they simply handle each gate individually.

Lemmas that support Theorem 1 are formally stated and proved in the full version of this paper.

Proofs of these lemmas follow from inspecting our recursive algorithms and (1) counting the number of calls to the underlying scheme's algorithms and (2) counting the number of garblings kept in scope.

We now draw attention to two key details of algorithms in Figure 3: (1) LS.Ev delegates to a subprocedure `EvCond` and (2) LS.Gb delegates to a subprocedure `GbCond`. All details of conditionals are handled by these two subprocedures. Aside from these delegations, the algorithms in Figure 3 are relatively unsurprising: the algorithms closely match [HK20a]'s construction and essentially provide infrastructure needed to host our contribution. We briefly discuss the most relevant details of these algorithms before returning to an extended discussion of `EvCond` and `GbCond` (c.f. Section 5.1):

- **Projectivity.** LogStack is a *projective garbling scheme* [BHR12]. Projectivity requires that the input *encoding string* e and output *decoding string* d have a specific format: they must both be a vector of pairs of labels such that the left element of each pair is a label encoding logical 0 and the right element of each pair is a label encoding 1. Thus, LS.En and LS.De are straightforward mappings between cleartext values and encoding/decoding strings.
- **Sequences and Translation.** In a sequence of two circuits, all output wires of the first circuit are passed as the inputs to the second. Because these two circuits are garbled starting from different seeds, the output labels from $\mathcal{C}_0$ will not match the required input encoding of $\mathcal{C}_1$. We thus implement a *translation* component ($trans$.Ev and $trans$.Gb) that implements via garbled rows a straightforward translation from one encoding to another. Our scheme securely implements the translator, and all other gadgets, using a PRF ([HK20a] used an RO). This simplification is possible because of the stronger property, strong stackability, that we require of the underlying garbling scheme (see Section 6).

5.1 Algorithms for Handling of Conditionals

With the remaining formalization out of the way, we focus on conditional branching. Our goal is to formalize `EvCond` and `GbCond`, the key sub-procedures invoked by LS.Ev and LS.Gb respectively. Our presentation is a formalization of discussion in Section 2; the following explores the technical aspects of our construction, but the reader should refer to Section 2 for unifying high level intuition.

Demultiplexer and Multiplexer. Before we discuss handling the body of conditionals, we briefly discuss entering and leaving a conditional. That is, we describe the *demultiplexer* (entry) and *multiplexer* (exit) components.

The demultiplexer is responsible for (1) forwarding the conditional's inputs to the active branch $\mathcal{C}_\alpha$ and (2) forwarding specially prepared garbage inputs to

each branch $\mathcal{C}_{i\neq\alpha}$. The demultiplexer computes the following function for each wire input x to each branch $\mathcal{C}_i$ with respect to the active index α:

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

where $\bot$ is a specially designated constant value. In the GC, the label corresponding to $\bot$ is independent yet indistinguishable from the corresponding 0 and 1 labels: independence is crucial for security. The demultiplexer is easily implemented by garbled rows. The number of required rows is proportional to the number of branches and the conditional's number of inputs. `EvCond` and `GbCond` make use of $demux$.Ev and $demux$.Gb, procedures which implement the above function via GC. Although we do not, for simplicity, formally describe these, we emphasize that they are a straightforward implementation of garbled rows.

The multiplexer is central to our approach. It non-interactively eliminates garbage outputs from inactive branches. Despite its central role, if `Gen` knows the garbage outputs from each branch, the multiplexer's implementation is simple. Specifically, suppose each branch $\mathcal{C}_i$ has an output x_i that should propagate if that branch is active. The multiplexer computes the following function:

$$mux(x_0, ..., x_{b-1}, \alpha) = x_\alpha$$

Given that (1) each value $x_{i\neq\alpha}$ is a fixed constant $\bot$, at least with respect to a given α (a property that we carefully arrange via the demultiplexer), and (2) `Gen` knows the value of each of these fixed constants (the central point of our work), then the above mux function is easily implemented as a collection of garbled rows. The number of required rows is proportional to the number of branches and the number of the conditional's outputs. `EvCond` and `GbCond` make use of mux.Ev and mux.Gb, procedures which implement the above function via GC. As with the demultiplexer, we do not formalize these procedures in detail, but their implementation is a straightforward handling of garbled rows.

Garbling Subtrees. Recall, we organize the b branches into a binary tree. For *each* internal node of the tree, both `EvCond` and `GbCond` perform a common task: they garble all branches in the entire subtree rooted at that node and stack together all material. These subtrees are garbled according to seeds given by the `SortingHat`, formally defined in Figure 2. Like the demultiplexer and multiplexer, the GC implementation of `SortingHat` is a straightforward handling of garbled rows: we assume procedures `SortingHat`.Ev and `SortingHat`.Gb which implement this handling.

We next define a procedure, `GbSubtreeFromSeed` (Figure 4), which performs the basic task of garbling and stacking an entire subtree. `GbSubtreeFromSeed` recursively descends through the subtree starting from its root, uses a PRF to derive child seeds from the parent seed, and at the leaves garbles the branches. As the recursion propagates back up the tree, the procedure stacks the branch materials together (and concatenates input/output encodings). The recursion

$\texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, i, j, seed)$:

  $\texttt{if}\ i = j$: ▷ Base case of 1 branch.

    $\texttt{return}\ \texttt{Gb}(\boldsymbol{\mathcal{C}}[i], seed)$

  $\texttt{else}$:

    ▷ Expand child seeds using PRF.

    $seed_L \leftarrow F_{seed}(0)$

    $seed_R \leftarrow F_{seed}(1)$

    ▷ Recursively garble both child trees and stack material.

    $k \leftarrow \texttt{halfway}(i, j)$

    $M_L, e_L, d_L \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, i, k, seed_L)$

    $M_R, e_R, d_R \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, k+1, j, seed_R)$

    $\texttt{return}\ (M_L \oplus M_R, e_L \mid e_R, d_L \mid d_R)$

$\texttt{halfway}(i, j)$:

  ▷ Simple helper for splitting range of branches (approximately) in half.

  $\texttt{return}\ i + \left\lfloor \frac{j-i}{2} \right\rfloor$

Fig. 4. The helper algorithm `GbSubtreeFromSeed` starts from a single seed at the root of a subtree $\mathcal{N}_{i,j}$, derives all seeds in the subtree, garbles all branches in the subtree, and stacks (using XOR) all resultant material. The procedure also returns the input/output encodings for all branches.

tracks two integers i and j, denoting the range of branches $\mathcal{C}_i..\mathcal{C}_j$ that are to be stacked together. `EvCond` and `GbCond` use a similar strategy, and all three algorithms maintain an invariant that i, j refers to a valid node $\mathcal{N}_{i,j}$ in the binary tree over the b branches. `EvCond` and `GbCond` invoke `GbSubtreeFromSeed` at *every* node. This entails that both procedures garble each branch $\mathcal{C}_i$ more than once, but with different seeds. As discussed in Section 2, this repeated garbling is key to reducing the total number of garbage outputs that `Eval` can compute.

Evaluating Conditionals. We now formalize the procedure `EvCond` by which `Eval` handles a vector of conditionals (Figure 5). The core of `EvCond` is delegated to a recursive subprocedure $\texttt{EvCond}'$. $\texttt{EvCond}'$ carefully manages material and uses the garblings of sibling subtrees to evaluate each branch while limiting the possible number of garbage outputs. $\texttt{EvCond}'$ is a formalization of the high level procedure described in Section 2: `Eval` recursively descends through the tree, constructing and unstacking garblings of subtrees in the general case. When she finally reaches the leaf nodes, she simply evaluates. In the base case $i = \alpha$, she will have correctly unstacked all material except M_α (because she has good seeds for the sibling roots of α), and hence evaluates correctly. All other cases $i \neq \alpha$ will lead to garbage outputs that `Gen` must also compute. Other than

$\texttt{EvCond}(\boldsymbol{\mathcal{C}}, M, X)$:

$b \leftarrow |\boldsymbol{\mathcal{C}}|$

▷ Parse the active branch index from the rest of the input.

$\alpha \mid X' \leftarrow X$

▷ Parse material for gadgets and body of conditional.

$M_{\texttt{SortingHat}} \mid M_{dem} \mid M_{cond} \mid M_{mux} \leftarrow M$

▷ Run `SortingHat` to compute all of `Eval`'s seeds.

$\texttt{es} \leftarrow \texttt{SortingHat.Ev}(\alpha, M_{\texttt{SortingHat}})$

▷ Run the demultiplexer to compute input for each branch $\mathcal{C}_i$.

$\boldsymbol{X}_{cond} \leftarrow demux.\mathsf{Ev}(\alpha, X, M_{dem})$

▷ We define a recursive subprocedure that evaluates $\mathcal{C}_i - \mathcal{C}_j$ using material M.

$\texttt{EvCond}'(i, j, M_{i,j})$:

`if` $i = j$:

▷ Base case: compute output by evaluating the branch normally.

▷ This base case corresponds to $\texttt{guess} = i$.

▷ Accumulate output labels into the vector $\boldsymbol{Y}_{cond}$ (for later garbage collection).

$\boldsymbol{Y}_{cond}[i] \leftarrow \mathsf{Ev}(\mathcal{C}_i, M, \boldsymbol{X}_{cond}[i])$

`else` :

$k \leftarrow \texttt{halfway}(i, j)$

▷ Garble the right subtree using the available seed,

▷ unstack, and recursively evaluate the left subtree.

$M_{k+1,j}, \cdot, \cdot \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, k+1, j, \texttt{es}_{k+1,j})$

$\texttt{EvCond}'(i, k, M_{i,j} \oplus M_{k+1,j})$

▷ Symmetrically evaluate the right subtree.

$M_{i,k}, \cdot, \cdot \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, i, k, \texttt{es}_{i,k})$

$\texttt{EvCond}'(k+1, j, M_{i,j} \oplus M_{i,k})$

▷ Start recursive process from the top of the tree.

$\texttt{EvCond}'(0, b-1, M_{cond})$

▷ Eliminate garbage and propagate $\boldsymbol{Y}_\alpha$ via the multiplexer.

`return` $mux.\mathsf{Ev}(\alpha, \boldsymbol{Y}_{cond}, M_{mux})$

Fig. 5. `Eval`'s procedure, `EvCond`, for evaluating a conditional with b branches. `EvCond` evaluates each branch; $b-1$ evaluations result in garbage outputs and one (the evaluation of $\mathcal{C}_\alpha$) results in valid outputs. The multiplexer collects garbage and propagates output from $\mathcal{C}_\alpha$. `EvCond` involves $b \log b$ calls to `Gb` (via `GbSubtreeFromSeed`), and each branch evaluation is done with respect to the garbling of that branch's sibling subtrees.

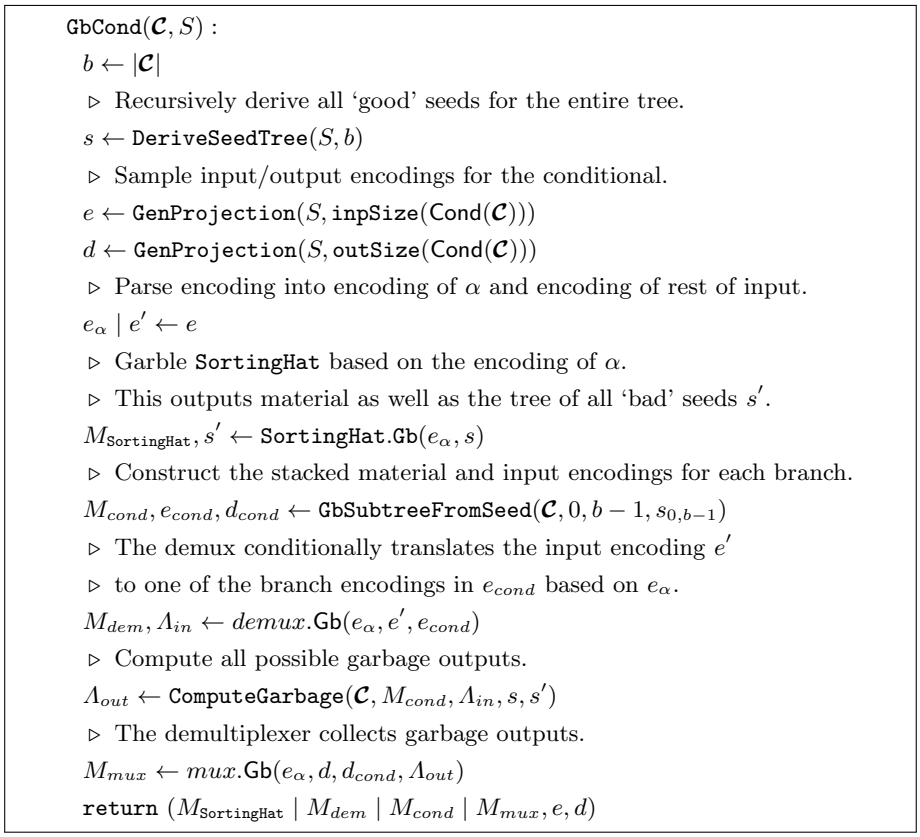

Fig. 6. The algorithm for garbling a conditional vector. Given b branches, GbCond returns (1) the stacked material, (2) the input encoding string, (3) all b output decoding strings, and (4) all $b \log b$ possible garbage output label vectors.

the delegation to EvCond′, EvCond simply invokes SortingHat.Ev to obtain her seeds, invokes $demux$.Ev to propagate valid inputs to $\mathcal{C}_\alpha$, and, after evaluating all branches, invokes mux.Ev to collect garbage outputs from all $\mathcal{C}_{i \neq \alpha}$.

Garbling Conditionals. Finally, we formalize Gen's procedure for handling vectors of conditional branches, GbCond (Figure 6).

1. GbCond recursively derives a binary tree of good seeds via DeriveSeedTree. This call uses a PRF to recursively derive seeds in the standard manner.
2. GbCond invokes GenProjection to select uniform input/output encodings e and d: e and d are vectors of pairs of labels that are the valid input/output labels for the overall conditional. Our use of GenProjection is straightforward and similar to that of [HK20a].
3. GbCond uses SortingHat.Gb to garble the SortingHat functionality of Figure 2. As input, GbCond provides the tree of good seeds s and the encoding

$\texttt{ComputeGarbage}(\boldsymbol{\mathcal{C}}, M, \Lambda_{in}, s, s')$:

▷ We first define a recursive subprocedure.

$\texttt{ComputeGarbage}'(i, j, M_{i,j}, \boldsymbol{M}')$:

▷ Compute all possible garbage outputs from branches $\mathcal{C}_i - \mathcal{C}_j$.

▷ $\boldsymbol{M}'$ is a vector of the bad garblings of all sibling roots of the current node.

`if` $i = j$:

▷ Base case: loop over all possible garbage material

▷ and accumulate garbage outputs into Λ_{out}.

$acc \leftarrow M_{i,i}$

`for` $k \in 0..|\boldsymbol{M}'| - 1$:

▷ Emulate all possible bad evaluations of $\mathcal{C}_i$.

$acc \leftarrow acc \oplus \boldsymbol{M}'[k]$

$\Lambda_{out}[i][k] \leftarrow \mathsf{Ev}(\boldsymbol{\mathcal{C}}[i], acc, \Lambda_{in}[k])$

`else` :

$k \leftarrow \texttt{halfway}(i, j)$

▷ Compute the good material for both subtrees.

$M_{i,k}, \cdot, \cdot \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, i, k, s_{i,k})$

$M_{k+1,j} \leftarrow M_{i,j} \oplus M_{i,k}$

▷ Compute the bad material for both subtrees.

$M'_{i,k}, \cdot, \cdot \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, i, k, s'_{i,k})$

$M'_{k+1,j}, \cdot, \cdot \leftarrow \texttt{GbSubtreeFromSeed}(\boldsymbol{\mathcal{C}}, k+1, j, s'_{k+1,j})$

▷ Recursively compute all garbage outputs.

$\texttt{ComputeGarbage}'(i, k, (M_{k+1,j} \oplus M'_{k+1,j}) \mid \boldsymbol{M}')$

$\texttt{ComputeGarbage}'(k+1, j, (M_{i,k} \oplus M'_{i,k}) \mid \boldsymbol{M}')$

$b \leftarrow |\boldsymbol{\mathcal{C}}|$

▷ Start the recursive process using the top level material M

▷ and using the empty vector of bad sibling material.

$\texttt{ComputeGarbage}'(0, b-1, M, [\,])$

`return` Λ_{out}

Fig. 7. `ComputeGarbage` allows `Gen` to compute the possible garbage output labels from evaluation of inactive branches. Specifically, the algorithm takes as arguments (1) the vector of conditional branches $\boldsymbol{\mathcal{C}}$, (2) the 'good' material for the conditional M, (3) the garbage input labels Λ_{in}, (4) the tree of 'good' seeds (i.e. the seeds used by `Gen` to generate M) s, and (5) the tree of 'bad' seeds s'. The algorithm outputs Λ_{out}, the vector (length b) of vectors (each length $\log b$) of output labels from each branch.

of the active branch id e_α. As output, `Gen` receives the tree of all bad seeds. `GbCond` needs these bad seeds, in addition to the good seeds he already knows, to emulate `Eval` making a bad guess.
4. `GbCond` uses `GbSubtreeFromSeed` to derive stacked material M_{cond} from the root seed. M_{cond} is the material that `Gen` ultimately sends to `Eval`.
5. `GbCond` calls $demux$.Gb to compute the demultiplexer garbled rows. This call also returns Λ_{in}, the collection of garbage input labels for each branch: essential information that allows `Gen` to emulate `Eval`.

With this accomplished, `GbCond`'s remaining task is to encrypt the garbage-collecting multiplexer. However, it is not clear how this can be achieved unless `Gen` *knows all garbage outputs* that `Eval` might compute. Thus, `GbCond` first invokes `ComputeGarbage` (Figure 7), a procedure which emulates all of `Eval`'s bad guesses.

`ComputeGarbage` delegates to the recursive subprocedure `ComputeGarbage'`. This recursive procedure walks down the tree, maintaining two key variables: (1) $M_{i,j}$ holds the correct material for the current subtree $\mathcal{N}_{i,j}$ and (2) $\boldsymbol{M'}$ holds a vector of bad materials of the incorrectly garbled sibling roots of $\mathcal{N}_{i,j}$. In the general case, these variables are simply appropriately updated via calls to `GbSubtreeFromSeed`. Thus, in the base case, the garbage materials for all sibling roots of the considered leaf are available. Additionally, all garbage inputs into each branch are available in the vector Λ_{in}. So, at the leaves we can compute all garbage outputs for each branch by calling Ev on the proper combinations of garbage material and labels. We store all garbage outputs into the global vector Λ_{out}, which is returned by the overall procedure, and then ultimately used by `GbCond` to call mux.Gb.

6 LogStack Correctness/Security

We discuss LogStack's correctness and security properties. We formalize our theorems in the [BHR12] framework (as modified by [HK20a]), which requires a candidate garbling scheme to be **correct**, **oblivious**, **private**, and **authentic**.

In addition, [HK20a] introduced a new property, **stackability**, which formalizes the class of garbling schemes whose garblings can be securely stacked; hence stackable schemes are candidate underlying schemes. In this work, we strengthen the definition of stackability. This strengthening, which we call **strong stackability**, allows us to prove security under standard assumptions (an improvement over [HK20a], which required a random oracle assumption). Strong stackability is strictly stronger than stackability: all strongly stackable schemes are stackable, and all lemmas that hold for stackable schemes hold also for strongly stackable schemes. A key application of this second fact is that all stackable schemes are trivially **oblivious**, so all strongly stackable schemes are oblivious. We prove security given a strongly stackable, correct, authentic, private underlying scheme.

[HK20a] showed that several standard garbling schemes are stackable, including the state-of-the-art half-gates technique [ZRE15]. We later argue that such schemes either are strongly stackable without modification or can be easily

adjusted. Hence, our implementation can assume an RO and use half-gates as its underlying scheme to achieve high performance.

LogStack is itself strongly stackable, giving flexibility in usage: while by design LogStack handles vectors of conditional branches, we also support arbitrarily nested conditional control flow without modifying the source program. We note that this nested usage *does not* give $O(b \log b)$ computation, and so vectorized branches should favored where possible.

Due to a lack of space, we postpone most proofs to the full version of this paper.

6.1 Correctness

Definition 1 (Correctness). *A garbling scheme is* ***correct*** *if for all circuits $\mathcal{C}$, all input strings $\boldsymbol{x}$ of length $\mathtt{inpSize}(\mathcal{C})$, and all pseudorandom seeds S:*

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

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

A correct scheme implements the semantics specified by ev. Proof of the following is formalized in the full version of this paper.

Theorem 2. *If Base is correct, then LogStack is correct.*

6.2 Security

The following definition is derived from the corresponding definition of [HK20a]; we discuss its motivation (support for PRF-based garbling gadgets) and technical differences with [HK20a] immediately after we present it formally below.

Definition 2 (Strong Stackability). *A scheme is* ***strongly stackable*** *if:*

1. *For all circuits $\mathcal{C}$ and all inputs $\boldsymbol{x}$,*

$$(\mathcal{C}, M, \mathsf{En}(e, \boldsymbol{x})) \stackrel{c}{=} (\mathcal{C}, M', \boldsymbol{X}')$$

   where S is uniformly drawn, $(M, e, \cdot) = \mathsf{Gb}(1^\kappa, \mathcal{C}, S)$, $\boldsymbol{X}' \leftarrow_\$ \{0,1\}^{|\boldsymbol{X}|}$, and $M' \leftarrow_\$ \{0,1\}^{|M|}$.
2. *The scheme is* ***projective*** *[BHR12].*
3. *There exists an efficient deterministic procedure* colorPart *that maps strings to $\{0,1\}$ such that for all $\mathcal{C}$ and all projective label pairs $A^0, A^1 \in d$:*

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

   where S is uniformly drawn and $(\cdot, \cdot, d) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C}\ S)$.
4. *There exists an efficient deterministic procedure* keyPart *that maps strings to $\{0,1\}^\kappa$ such that for all $\mathcal{C}$ and all projective label pairs $A^0, A^1 \in d$:*

$$\mathsf{keyPart}(A^0) \mid \mathsf{keyPart}(A^1) \stackrel{c}{=} \{0,1\}^{2\kappa}$$

   where S is uniformly drawn and $(\cdot, \cdot, d) \leftarrow \mathsf{Gb}(1^\kappa, \mathcal{C}\ S)$.

The above definition is given by [HK20a], with the exception of point 4. Informally, stackability ensures (a) that circuit garblings 'look random' and (b) that our scheme can manipulate labels generated by the underlying scheme. Since strong stackability simply adds point 4, the following lemma is immediate:

Lemma 1. *Every strongly stackable scheme is stackable.*

We briefly explain the role of $\mathsf{colorPart}$ and $\mathsf{keyPart}$. As with [HK20a], we use the output labels of the underlying scheme as keys in subsequent garbled gadgets. The $\mathsf{keyPart}$ procedure allows us to extract a *suitable PRF key* from each label. At the same time, we make use of the classic point-and-permute trick to reduce the number of PRF calls needed to evaluate garbled gadgets: we use the $\mathsf{colorPart}$ as the bit that instructs which garbled row to decrypt. Note that because we essentially 'split' each output label into a key and a color, we 'lose' bits of the underlying scheme's labels when we invoke $\mathsf{keyPart}$. We stress that this is not an issue: the required key length for the next PRF application can be restored as we require $\mathsf{keyPart}$ output to be κ bits long. All point-and-permute schemes have a similar approach.

The added requirement (point 4) allows us to relax our security assumptions in comparison to [HK20a]. For each projective output pair A^0, A^1, we require that $\mathsf{keyPart}(A^0)$ and $\mathsf{keyPart}(A^1)$ are unrelated. This is achieved by requiring that the concatenation of these two strings is indistinguishable from a random string of the same length. This allows us to circumvent a problem: the [HK20a] definition allowed labels in the underlying scheme to be arbitrarily related. More precisely, while point 1 requires that any particular set of labels seen by `Eval` look random, it does not require that *all labels together* look random. This was problematic, because the output labels of the underlying scheme were used to implement garbled tables, so the two possibly related labels were both used as PRF keys. Using related keys is outside the scope of the standard PRF security definition. Thus, [HK20a] were forced to assume the existence of a random oracle to ensure possible relationships in the output decoding string did not compromise security. By adding point 4, we ensure that the *entire* decoding string 'looks random', so all labels must be independent. This added requirement on the underlying scheme allows us to push our proofs through in the standard model.

Many standard schemes are compatible with strong stackability: if the scheme is stackable and has randomly chosen output labels, it trivially satisfies our definition. Free XOR based schemes [KS08] use pairs of labels separated by a fixed constant Δ, and so are not *a priori* strongly stackable. However, it is easy to adjust such schemes such that the final output gates return independent labels. As a final note, while our scheme is secure in the standard model, we of course adopt any additional security assumptions from the chosen underlying scheme: e.g., instantiating $\mathsf{LogStack}$ with the efficient Half Gates scheme [ZRE15] requires us to assume the existence of a circular correlation robust hash function.

We prove the following in the full version of this paper. The proof utilizes properties of Base and of a PRF to show that $\mathsf{LogStack}$'s garblings 'looks random'.

Theorem 3. *If* Base *is strongly stackable, then* LogStack *is strongly stackable.*

Definition 3 (Obliviousness). *A garbling scheme is* ***oblivious*** *if there exists a simulator $\mathcal{S}_{obv}$ such that for any circuit $\mathcal{C}$ and all inputs $\boldsymbol{x}$ of length* $\texttt{inpSize}(\mathcal{C})$*, the following are indistinguishable:*

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

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

Obliviousness ensures that the garbled circuit with input labels can be simulated, and hence reveals no extra information to `Eval`. [HK20a] proved that every stackable scheme is trivially oblivious: drawing a random string of the correct length is a suitable simulator. This fact, combined with Lemma 1 and Theorem 3 implies two immediate facts:

Lemma 2. *Every strongly stackable scheme is oblivious.*

Theorem 4. *If* Base *is strongly stackable, then* LogStack *is oblivious.*

Definition 4 (Authenticity). *A garbling scheme is* ***authentic*** *if for all circuits $\mathcal{C}$, all inputs $\boldsymbol{x}$ of length* $\texttt{inpSize}(\mathcal{C})$*, and all poly-time adversaries $\mathcal{A}$ the following probability is negligible in κ:*

$$\mathsf{Pr}(\boldsymbol{Y}' \neq \mathsf{Ev}(\mathcal{C}, M, \boldsymbol{X}) \wedge \mathsf{De}(d, \boldsymbol{Y}') \neq \bot)$$

where S is uniform, $(M, e, d) = \mathsf{Gb}(1^\kappa, \mathcal{C}, S)$, $\boldsymbol{X} = \mathsf{En}(e, \boldsymbol{x})$, and $\boldsymbol{Y}' = \mathcal{A}(\mathcal{C}, M, \boldsymbol{X})$

Authenticity ensures that an adversary cannot compute GC output labels except by running the scheme as intended.

We prove the following in the full version of this paper. The proof utilizes properties of Base and of a PRF to show that an adversary cannot compute GC output labels except by running LogStack.

Theorem 5. *If* Base *is authentic, then* LogStack *is authentic.*

Definition 5 (Privacy). *A garbling scheme is* ***private*** *if there exists a simulator $\mathcal{S}_{prv}$ such that for any circuit $\mathcal{C}$ and all inputs $\boldsymbol{x}$ of length* $\texttt{inpSize}(\mathcal{C})$*, the following are computationally indistinguishable:*

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

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

Privacy ensures that `Eval`, who is given access to $(M, \boldsymbol{X}, d)$, learns nothing except what can be learned from the output $\boldsymbol{y}$. I.e., `Gen`'s input is protected.

We prove the following in the full version of this paper. The proof utilizes properties of Base and of a PRF to show that `Eval`'s view can be simulated.

Theorem 6. *If* Base *is private, authentic, and strongly stackable, then* LogStack *is private.*

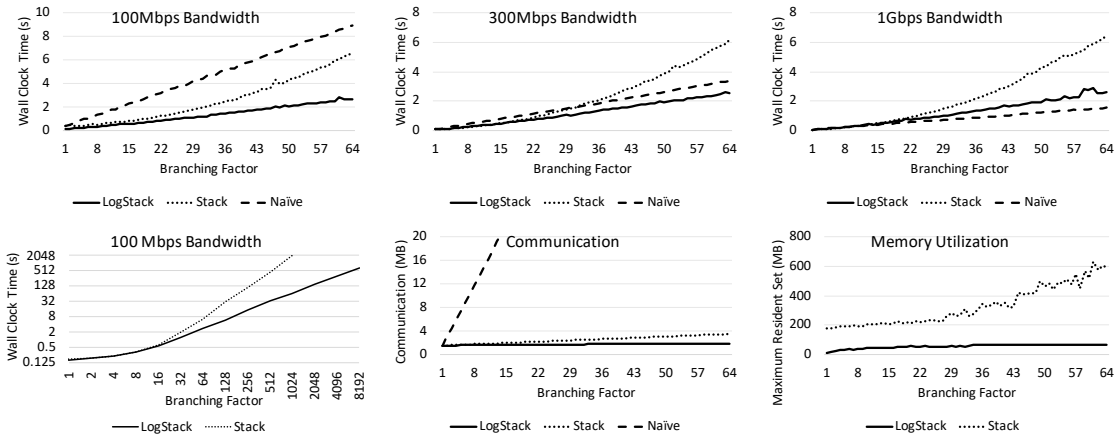

Fig. 8. Experimental evaluation of LogStack as compared to [HK20a]'s Stack and to basic half-gates [ZRE15] ('naïve' branching). We compare in terms of wall-clock time on different simulated network bandwidths (top). We performed an extended wall-clock time comparison to Stack (bottom left). Both LogStack and Stack greatly outperform basic half-gates in terms of total bandwidth consumption (bottom center), and LogStack greatly outperforms Stack in terms of memory consumption (bottom right).

7 Instantiation and Experimental Evaluation

We implemented LS in ~ 1500 lines of C++ and used it to instantiate a semihonest 2PC protocol. We instantiated Base using the half-gates [ZRE15], allowing high concrete performance. Our implementation thus relies on non-standard assumptions. We use computational security parameter $\kappa = 127$; the 128th bit is reserved for point and permute. Our implementation spawns additional threads to make use of inherent parallelism available in `GbCond` and `EvCond`.

Our experiments were each performed on a MacBook Pro laptop with an Intel Dual-Core i5 3.1 GHz processor and 8GB of RAM.

We compared our implementation to basic half-gates [ZRE15] and to the Stack SGC of [HK20a]. Figure 8 plots the results of our experiments.

We consider end-to-end wall-clock time, bandwidth consumption, and memory utilization. All branches implement the SHA-256 netlist, which has 47726 AND gates, 179584 XOR gates, and 70666 NOT gates. A GC for each branch has size 1.45 MB. It is, of course, unrealistic that a conditional would have the same circuit in each branch. However, we choose this benchmark because SHA-256 has become somewhat of a community standard and because our goal is only to analyze performance. We ensure our implementation does not cheat: it cannot recognize that branches are the same and hence cannot shortcut the evaluation.

Bandwidth consumption is the easiest metric to analyze. The communication chart in Figure 8 plots communication as a function of branching factor. As expected, Stack's and LogStack's communication remains almost constant, while

half-gates' grows linearly and immediately dominates. LogStack is slightly leaner than Stack because of low-level improvements to LogStack's demultiplexer. This small improvement should not be counted as a significant advantage over Stack.

Memory utilization was measured as a function of branching factor. We compare our scheme to Stack (half-gates memory utilization is constant, since garblings can be streamed across the network and immediately discarded). Our chart shows Stack's linear and LogStack's logarithmic space consumption. In settings with many branches, improved space consumption is essential. For example, we ran LogStack on a circuit with 8192 SHA-256 branches, a circuit that has $>$ 385M AND gates. Our peak memory usage was $\sim$ 100MB, while [HK20a] would require more than 12GB of space to run this experiment.

Wall-clock time to complete an end-to-end 2PC protocol is our most comprehensive metric. We plot three charts for 1 to 64 branches (on networks with 100, 300, and 1000 Mbps bandwidth) comparing each of the three approaches. We also explored more extreme branching factors, running conditionals with branching factors at every power of 2 from 2^0 to 2^{13} in the 100Mbps setting.

In the 1Gbps network setting, as expected, naïve half-gates leads. As discussed in Section 1.5, two cores (our laptop) indeed cannot keep up with the available network capacity. However, doubling the number of cores would already put us ahead of naïve, and any further computation boost would correspondingly improve our advantage. We are about $3\times$ faster than Stack.

In the 300Mbps network setting, we outperform naïve. Because we range over the same number of branches, we are the same factor $\approx 3\times$ faster than Stack.

The more typical 100Mbps setting shows the advantage of SGC. Both Stack and LogStack handily beat naïve.

Finally, we experimented with large branching factors. LogStack scales well; we ran up to 8192 branches as it was sufficient to show a trend. Due to its logarithmic memory utilization, LogStack would run on a practically arbitrary number of branches. In contrast, Stack exhibited limited scaling. We ran up to 1024 branches with Stack, enough to show a trend, and after which our experiments started to take too long. LogStack ran 2PC for a 1024-branch conditional in $\sim 67s$, while Stack took $\sim 2050s$, $\sim 31\times$ slower than LogStack.

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

References

[AGKS20] Masaud Y. Alhassan, Daniel Günther, Ágnes Kiss, and Thomas Schneider. Efficient and scalable universal circuits. *Journal of Cryptology*, 33(3):1216–1271, July 2020.

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

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

[GKS17] Daniel Günther, Ágnes Kiss, and Thomas Schneider. More efficient universal circuit constructions. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 443–470. Springer, Heidelberg, December 2017.

[GLNP15] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 567–578. ACM Press, October 2015.

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

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

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

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

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

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

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

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

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

[ZYZL19] Shuoyao Zhao, Yu Yu, Jiang Zhang, and Hanlin Liu. Valiant's universal circuits revisited: An overall improvement and a lower bound. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 401–425. Springer, Heidelberg, December 2019.