One Hot Garbling

David Heath
Georgia Tech
USA
heath.davidanthony@gatech.edu

Vladimir Kolesnikov
Georgia Tech
USA
kolesnikov@gatech.edu

ABSTRACT

Garbled Circuit (GC) is the main practical 2PC technique, yet despite great interest in its performance, GC notoriously resists improvement. Essentially, we only know how to evaluate GC functions gate-by-gate using encrypted truth tables; given input labels, the GC evaluator decrypts the corresponding output label.

Interactive protocols enjoy more sophisticated techniques. For example, we can expose to a party a (masked) private value. The party can then perform useful local computation and feed the resulting cleartext value back into the MPC. Such techniques are not known to work for GC.

We show that it is, in fact, possible to improve GC efficiency, while keeping its round complexity, by exposing masked private values to the evaluator. Our improvements use garbled *one-hot* encodings of values. By using this encoding we improve a number of interesting functions, e.g., matrix multiplication, integer multiplication, field element multiplication, field inverses and AES S-Boxes, integer exponents, and more. We systematize our approach by providing a framework for designing such GC modules.

Our constructions are concretely efficient. E.g., we improve binary matrix multiplication inside GC by more than 6× in terms of communication and by more than 4× in terms of WAN wall-clock time.

Our improvement *circumvents* an important GC lower bound and may *open* GC to further improvement.

CCS CONCEPTS

• Security and privacy → Cryptography; **• Theory of computation → Cryptographic protocols**.

KEYWORDS

Secure 2PC; Garbled Circuits

ACM Reference Format:
David Heath and Vladimir Kolesnikov. 2021. One Hot Garbling. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3460120.3484764

1 INTRODUCTION

Garbled circuits (GCs) allow two mutually untrusting parties to compute arbitrary functions of their private inputs while revealing only the functions' outputs. Today, cryptographers view GC as a cryptographic primitive rather than a protocol. The primitive can be plugged into many protocols and is foundational in secure multiparty computation (MPC).

The GC primitive only allows the circuit generator G and evaluator E to communicate a constant number of times. This restriction makes GC difficult to improve. Indeed, since Yao first described GC, only a handful of fundamental improvements have been made.

GCs are usually structured as encryptions of Boolean circuits composed of XOR and AND gates. Most prior work has focused on improving these gates. The most relevant cost is bandwidth consumption: G must send to E an 'encryption' of the circuit, and this transmission is typically understood to be the GC bottleneck.

The widely used half-gates [ZRE15] garbling requires G send to E two ciphertexts per AND gate; XOR gates are communication free [KS08]. [ZRE15] also established a matching lower bound on AND gate communication that is difficult to circumvent[1].

Thus, it is natural to target GC evaluation of more complex functions. Thus, searching for complex functions that can be quickly computed in GC became a natural research direction. Nevertheless, only two core-GC improvements have subsequently been found:

Arithmetic GCs [BMR16] show that Free XOR [KS08] can be generalized to achieve free additive homomorphisms for *arbitrary* fields. Using this technique, we can efficiently add arithmetic values inside GC. Unfortunately, both multiplying field values and converting between fields is expensive, since these operations require G to send to E a number of ciphertexts proportional to the size of the field. Thus, arithmetic GCs only improve communication in very specific scenarios.

Stacked garbling [HK20a] improves the performance of GCs for functions that include conditional branching. The technique shows that G needs to send a number of ciphertexts proportional to only the longest program execution path, not to the entire circuit. Stacked garbling dramatically improves some functions, but requires that these functions feature exclusive conditional behavior.

Our work. We show that a number of useful functions can be greatly improved by operating over a garbled *one-hot* encoding.

Specifically, suppose the GC holds two bit vectors $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$. Moreover, suppose E knows a in cleartext. Our central primitive allows G and E to compute the following $2^n \times m$

[1] "Three-halves" [RR21], a recent improvement to half gates developed concurrently with our work, requires only 1.5 ciphertexts per AND gate. This new approach circumvents the letter, but not the spirit, of the [ZRE15] lower bound by operating on portions of garbled labels. The core of the lower bound still holds and implies further improvements will be difficult.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484764

Application	**Comm. Improvement**
128×128 binary matrix mult.	6.2×
32-bit mult.	1.5×
GF(2^8) mult.	2.2×
AES S-Box	1.1×
32-bit x^y for public x	11.8×
32-bit x mod p for public p	3.3×

Figure 1: Use cases that we implemented where a one-hot encoding improves over a standard Boolean circuit implemented with [ZRE15]. We list communication reduction as compared to a standard circuit. See Section 7 for details.

matrix inside the GC extremely efficiently:

$$\begin{bmatrix} 0 & 0 & \cdots & 0 \\ & & \vdots & \\ 0 & 0 & \cdots & 0 \\ b_0 & b_1 & \cdots & b_{m-1} \\ 0 & 0 & \cdots & 0 \\ & & \vdots & \\ 0 & 0 & \cdots & 0 \end{bmatrix} \tag{1}$$

In this matrix, row a, viewed as $a \in \{0, 2^{n-1}\}$, is the only non-zero row.

At first glance, this primitive, which we call a one-hot outer product, may seem incredibly contrived and niche. It is not.

This primitive can be used to implement a number of important functions. We use it to improve the GC bandwidth consumption of matrix multiplication, integer multiplication, field multiplication, field inverses and AES S-Boxes, integer exponents, and more. We believe other efficient applications of the technique are likely.

We develop a framework for designing such ready-to-use modules. Once designed, these modules are freely *composable* in GC.

1.1 Contribution

Non-interactivity is a key advantage of GC, as compared to other MPC techniques, such as GMW. However, non-interactivity also severely limits the set of GC building blocks. Essentially, we only know how to evaluate GC functions by using encrypted truth tables; given input labels, E decrypts the corresponding output label.

In this work, we show that it is possible to improve GC efficiency by exposing masked private values to E. By doing so, we *circumvent* the [ZRE15] GC lower bound, and *open* GC for further improvement. In more detail, we:

(1) Introduce a new GC gate primitive that computes a one-hot outer product (see Equation (1)) for only $2(n-1)+m$ ciphertexts.
(2) Provide numerous constructions that utilize this new primitive to implement improved GC modules (see Figure 1 and Section 7).
(3) Formalize a *framework* that allows new one-hot-based modules to be easily plugged in. Once implemented, these new modules can be used as if they are ordinary gates.
(4) Implement our approach in C++ and provide experimental evaluation (see Section 7).

1.2 High Level Intuition

Let $\mathcal{H}(\cdot)$ denote the function that maps a bit vector to its corresponding *one-hot* encoding. That is, for $a \in \{0,1\}^n$, $\mathcal{H}(a)$ is a length-2^n bit vector that is zero everywhere, except at index a, where it is one. Let $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ be two bit vectors and suppose E holds garblings of these two values. Our lowest level primitive allows E to efficiently construct a garbling of the following matrix (see also Equation (1)):

$$\mathcal{H}(a) \otimes b$$

where $\otimes$ denotes the vector outer product operation. This matrix has dimension $2^n \times m$, yet the parties construct the output using only $O(n+m)\kappa$ bits of communication, for security parameter κ.

Our construction does have one limitation: E must know a. Nevertheless, we build a number of useful GC constructions from this low-level primitive, even if E does *not* know the input.

Our constructions use two key ideas:

First, the garbled one-hot encoding of a value is, in a sense, 'fully homomorphic'. Namely, let $\mathcal{T}(f)$ denote the *truth table* (represented as a matrix) for arbitrary function f. The following equality holds:

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

Thus, if f is public and E knows a, then we can map a garbling of $\mathcal{H}(a)$ to a garbling of $f(a)$ *without communication.* Specifically, the parties locally construct and apply $\mathcal{T}(f)^{\intercal}$ via Free XOR [KS08].

Second, we can reveal in cleartext to E *masked* intermediate circuit values. This way, E learns nothing, yet can use the above one-hot primitive to compute f of masked a. In many useful cases we can use simple algebra to cheaply undo the masking and obtain $f(a)$ inside GC, where E does *not* know a.

2 RELATED WORK

Ours is in a line of works that improve the practical performance of GC. We review other works in this line. Our emphasis is *communication reduction*, which is the GC bottleneck.

Practical GC research has long focused on efficient evaluation of AND/XOR gates. [NPS99] gave the first GC communication improvement: garbled row reduction. Much later [KS08], gave the important Free XOR optimization which eliminated the communication cost of XOR gates. Garbled gates were slowly improved [PSSW09, KMR14], and the half-gates technique [ZRE15] reduced AND gate communication cost to only two ciphertexts. Subsequently, [GLNP18] showed that similar costs (two ciphertexts per AND and one per XOR) are possible even when assuming only one-way functions, as opposed a circular correlation robust hash function. Very recently, a new "three-halves" garbling technique showed that only 1.5 ciphertexts are needed per AND gate [RR21]. However, "three-halves" has not yet been implemented, so we focus our comparison on the widely-available half-gates technique. We mention that [RR21]'s construction uses Free XOR based GC labels, and so is compatible with and complementary to our technique: one-hot gates can be composed with 1.5 ciphertext AND gates in a single circuit.

Not only have there been *few* core-GC improvements, but those improvements have also been *small.* For example, half-gates improved over the prior state-of-the-art [KMR14] by only $1-1.5\times$ (1.5× is for the pessimal case where [KMR14]'s heuristics fail).

Recently, GC performance improvement has proceeded in two new directions:

(1) *Consider more expressive fields.* One elegant direction views a circuit as an object that operates over the Boolean *field*. With this perspective, it is natural to consider whether larger finite fields are also candidates for GC evaluation. [BMR16] showed that they indeed are candidates, and gave constructions that add/multiply in a small arithmetic field and even that convert between different fields. Unfortunately, multiplication/conversion gates grow linearly in the *size* of the considered fields, and so rapidly become impractical. Arithmetic GCs are, unfortunately, only useful in specific settings.
(2) *Consider more expressive functions.* Since an improved AND gate seems unlikely, it is natural to consider more complex functions. However, such improvements are elusive in GC, and, to our knowledge, only one has been made: stacked garbling [HK20a, HK21] shows that GC communication can be improved for functions with exclusive conditional behavior.

Our work falls into the second category.

Non-GC Expressive Functions. In other MPC protocols, it is possible to improve beyond considering simple XOR/AND or ADD/MUL gates. For example, other protocols allow (1) efficient lookup-table-based approaches [IKM$^+$13, DKS$^+$17, KKW17, DNNR17][2], (2) efficient linear algebra operations, e.g. [HKP20, ADI$^+$17, PSSY20, RWT$^+$18], or (3) custom designed subprotocols, such as fast field inverse computation [BIB89]. Our work brings a flavor of such techniques to GC.

Puncturable PRFs and MPC. Our one-hot outer product construction uses a well-known puncturable PRF derived from the classic GGM PRF [GGM84]. This same idea is often applied in MPC, for example to help achieve efficient OT extension [BCG$^+$19, YWL$^+$20]. Our work shows that this primitive can be directly and elegantly plugged into GC and that the resulting primitive is powerful.

GC frameworks. Part of our contribution is a *framework* for building new GC modules from one-hot outer products. The generally accepted GC framework, specified by [BHR12], defines garbling schemes. We clarify that our framework and [BHR12]'s framework achieve different goals. The [BHR12] framework provides an abstraction barrier between high level protocols and garbling schemes. Our framework instead supports new GC modules which are hosted inside a specific garbling scheme. Indeed, our framework is proved secure in the [BHR12] framework.

Previous work, e.g., [KNR$^+$17, GLMY16] viewed their circuits as modules. The similarity between these works and ours is superficial. They build modular GC components that are individually garbled, then dynamically stitched together into a full GC for improved performance. In contrast, our modules enforce scope of private variables, and facilitate clean security proofs of circuits composed of our one-hot gates.

3 NOTATION AND ASSUMPTIONS

We list some simple notation here. We elaborate on more involved notation in the following subsections.

- κ is our computational security parameter, e.g. 128.
- G is the GC generator. We refer to G by he/him.
- E is the GC evaluator. We refer to E by she/her.
- $x \triangleq y$ denotes that x is equal to y by definition.
- $x \overset{c}{=} y$ denotes that x is computationally indistinguishable from y.
- We work with vectors and matrices:
  - If v is a vector, then v_i denotes the ith entry in v. If m is a matrix, then $m_{i,j}$, denotes the entry at the ith row and jth column. We use zero-based indexing.
  - m^T denotes the transpose of m.
  - $x \otimes y$ denotes the *outer product* of vectors x and y. The outer product can be defined as follows: $x \otimes y \triangleq x \cdot y^\mathsf{T}$.
- Let $\mathcal{D}$ be a distribution. We write $x \longleftarrow \mathcal{D}$ to denote that x is drawn from $\mathcal{D}$.
- We *overload* the notion of a circuit wire to hold a matrix of bits of arbitrary dimension. We sample wires in a natural manner from general $\mathcal{D}$. Namely, we sample $\mathcal{D}$, encode the result in binary, then store the result onto the wires.
- $[n]$ denotes the sequence of natural numbers $0, 1, ..., n-1$.

3.1 One-Hot Encoding and Truth Tables

Recall from Section 1.2 that our central construction computes the one-hot outer product $\mathcal{H}(a) \otimes b$. Moreover, we apply functions to one-hot encodings via truth tables. We define appropriate notation:

Definition 3.1 (One-hot encoding). Let $a \in \{0,1\}^n$ be a length-n bitstring. The *one-hot encoding* of a is a length-2^n bitstring denoted $\mathcal{H}(a)$ such that for all $i \in [n]$:

$$\mathcal{H}(a)_i \triangleq \begin{cases} 1 & \text{if i} = a \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.2 (Truth table). Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function. The *truth table* for f, denoted $\mathcal{T}(f)$, is a $2^n \times m$ matrix of bits such that:

$$\mathcal{T}(f)_{i,j} \triangleq f(i)_j$$

That is, the ith row of $\mathcal{T}(f)$ is the bitstring $f(i)$.

We extensively use the following simple lemma that relates truth tables and one-hot encodings:

Lemma 3.3 (Evaluation by truth table). *Let $f : \{0,1\}^n \to \{0,1\}^m$ be an arbitrary function. Let $a \in \{0,1\}^n$ be a bitstring:*

$$\mathcal{T}(f)^\mathsf{T} \cdot \mathcal{H}(a) = f(a)$$

Proof. Straightforward from Definitions 3.1 and 3.2. Informally, the one-hot vector "selects" row a of the truth table. □

3.2 GC Notation: Garbled Sharings

In this work, we forgo the standard GC notation of garbled labels in favor of *garbled sharings* of cleartext values held by G and E. This will be convenient for handling vectors and matrices of bits. We stress that the GC mechanism, including communication rounds, remains completely unchanged.

[2] Technically, large lookup tables can be implemented in GC by enumerating garbled rows, but this is expensive.

We use Free XOR style garbled circuit labels [KS08]. In the GC, G and E hold *sharings* of each circuit wire. Each sharing consists of two shares, one held by G and one by E. G samples a uniform value $\Delta \in \{0,1\}^\kappa$; Δ is a value that is *global* to all wires in the circuit. Then, for each wire value $a \in \{0,1\}$, G samples a uniform value $A \in \{0,1\}^\kappa$. A is G's share; E holds $A \oplus a\Delta$. Hence, the two parties together hold an XOR share of $a\Delta$. We will say that gates "output" a sharing. This corresponds to the traditional notation of E obtaining a valid wire label which can be used in continued GC evaluation.

Definition 3.4 (Garbled sharing). Let $a \in \{0,1\}$ be a bit. Let $A, B \in \{0,1\}^\kappa$ be two bitstrings. We say that the pair (A, B) is a *garbled sharing* of a over (usually implicit) $\Delta \in \{0,1\}^\kappa$ if $A \oplus B = a\Delta$. We denote a garbled sharing of a by writing $[\![a]\!]$:

$$[\![a]\!] \triangleq (A, B) \qquad \text{such that } A \oplus B = a\Delta$$

Each of the two elements in the sharing are called *shares*. In the GC, G holds one share and E holds the other. We say that a garbled sharing is *uniform* if one share is drawn uniformly from $\{0,1\}^\kappa$.

We extend sharing notation to vectors/matrices: a sharing of a matrix is a matrix of sharings. I.e., for a matrix $a \in \{0,1\}^{n\times m}$:

$$[\![a]\!] = \left[\!\!\left[\begin{matrix} a_{0,0} & \cdots & a_{0,m-1} \\ \vdots & \ddots & \\ a_{n-1,0} & & a_{n-1,m-1}\end{matrix}\right]\!\!\right] \triangleq \begin{bmatrix} [\![a_{0,0}]\!] & \cdots & [\![a_{0,m-1}]\!] \\ \vdots & \ddots & \\ [\![a_{n-1,0}]\!] & & [\![a_{n-1,m-1}]\!]\end{bmatrix}$$

Note, XOR is homomorphic over garbled sharings [KS08]:

$$[\![a]\!] \oplus [\![b]\!] = [\![a \oplus b]\!]$$

More generally, we can homomorphically apply arbitrary linear functions to sharings. Specifically, if f is a linear map, then we overload function application syntax as follows:

$$f([\![a]\!]) = f((A, A \oplus a\Delta)) \triangleq (f(A), f(A \oplus a\Delta))$$

That is, the parties apply (linear) f to a sharing by locally applying f to their respective shares. This generates a correct output sharing:

Lemma 3.5. *Let f be a linear map and let $[\![a]\!]$ be a sharing. Then*

$$f([\![a]\!]) = [\![f(a)]\!]$$

Proof.

$$\begin{aligned} & f([\![a]\!]) \\ &= f((A, A \oplus a\Delta)) && \text{Definition 3.4} \\ &= (f(A), f(A \oplus a\Delta)) && \text{function application to sharing} \\ &= (f(A), f(A) \oplus f(a)\Delta) && f \text{ is a linear map} \\ &= [\![f(a)]\!] && \text{Definition 3.4} \quad \square \end{aligned}$$

We apply the above fact often, most notably when applying truth tables to shared one-hot vectors. Specifically for *arbitrary* function f, Lemma 3.3 and Lemma 3.5 together imply the following:

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

3.2.1 G constants. It is easy for G to inject secret constants into the GC. Specifically to input a constant c, E takes as her share 0 and G takes $c\Delta$: note that this matches Definition 3.4. We use such constants to help eliminate introduced masks.

3.2.2 Share colors. GC techniques use garbled shares to decrypt ciphertexts arranged in tables. The classic point and permute technique [BMR90] shows that E need not try to decrypt each row of a table, but rather can use share "pointer bits" to directly decrypt the appropriate row. Per [ZRE15], we refer to these pointers as *colors*.

Namely, each share has a single distinguished bit that we refer to as the color. The key property is that on each wire, E's two possible shares have different colors, and the color of a share is independent of the cleartext value that the share represents.

Formally, we ensure that the global value Δ has a one in its least significant bit. We define a procedure Color that, when given a bit sharing $[\![a]\!]$, returns to G and E the least significant bit of their respective shares. Note the following:

$$\begin{aligned} &\text{Color}([\![a]\!]) = \text{Color}((A, A \oplus a\Delta)) \\ &= (\text{Color}(A), \text{Color}(A \oplus a\Delta)) = (\text{Color}(A), \text{Color}(A) \oplus a) \end{aligned}$$

That is, if both parties compute the color of their respective shares, the result is an XOR secret share of the cleartext value. We extend the Color procedure over vectors and matrices: the color of a matrix of sharings is the matrix of colors of its elements.

3.3 Model and Cryptographic Assumptions

We use the Free XOR technique [KS08] and so we assume a circular correlation robust hash function H [CKKZ12]. In practice, we can instantiate H using fixed-key AES [GKWY20, BHKR13].

Formally, we construct a garbling scheme [BHR12], which is a tuple of algorithms that can be plugged into GC protocols. Thus we do not need to formally consider a specific threat model, e.g. semi-honest adversaries. Informally, E and G can be understood as semi-honest. Our implementation (see Section 6) uses our garbling scheme to instantiate a semi-honest protocol.

4 TECHNICAL OVERVIEW

In this section, we present our techniques with sufficient detail to understand our contribution. Section 5 later presents our constructions in formal detail with appropriate theorems and proofs, and Section 7 shows a number of interesting functions that can be computed efficiently from our technique.

Let $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ be two strings. Recall from Section 1.2 that our core primitive efficiently computes the following:

$$[\![a]\!], [\![b]\!] \mapsto [\![\mathcal{H}(a) \otimes b]\!]$$

To use the primitive, E must know in cleartext the value a. We first sketch the construction, then show how it can be used.

4.1 Garbled One-Hot Encoding $[\![a]\!] \mapsto [\![\mathcal{H}(a)]\!]$

We first describe how to compute $[\![a]\!] \mapsto [\![\mathcal{H}(a)]\!]$ when E knows a. The idea marries GC with a well-known puncturable PRF built from the classic GGM PRF [GGM84]. Puncturable PRFs are useful in a number of settings, see e.g. [BW13, KPTZ13, BGI14, Ds17, BCG+19, SGRR19]. The technique is well known, but we nevertheless sketch it here and emphasize its natural compatibility with GC sharings.

G first generates a full binary tree of PRG seeds with 2^n leaves in the natural manner. Namely, each node's seed is derived by evaluating a PRG on its parent's seed. Let $S_{i,j}$ denote the jth seed on level i. Let the root of the overall tree reside in level -1. Let L_j be a pseudonym for the jth leaf seed: $L_j \triangleq S_{n-1,j}$.

Our goal is to deliver to E all leaf seeds $L_{j \neq a}$. Recall that G and E hold garbled shares of $[\![a]\!]$. Let $[\![a_i]\!] = (A_i, A_i \oplus a_i\Delta)$ be the shares of the individual bits in a. Recall, E knows each a_i in cleartext but *does not* know Δ. We can use these shares to encrypt values that help E recover each seed in the binary tree, except the seeds along the path to L_a.

As a base case, G simply defines the seeds on level zero as follows:

$$S_{0,0} \triangleq A_0 \oplus \Delta \quad S_{0,1} \triangleq A_0$$

Thus, E trivially obtains exactly one seed on level zero.

Now, consider arbitrary level i. Assume E has all seeds on level i except for one (along the path to L_a). By applying a PRG to these seeds, E can recover all seeds in level $i+1$ save two.

To deliver to E the missing seed "just off" the path to L_a, G sends two encrypted values. Let Even (resp. Odd) denote the XOR sum of all seeds $S_{i+1,j}$ for even j (resp. for odd j). G sends to E Even encrypted by $A_i \oplus \Delta$ and Odd encrypted by A_i. Thus, E can decrypt Even if the seed just off the path to L_a is even (resp. for odd). E can then XOR in the even seeds (resp. odd seeds) she already holds and recover the missing seed.

G now holds each seed L_i and E holds each $L_{i \neq a}$. By Definition 3.4, the parties hold garbled sharings of zero at all points $i \neq a$. To complete the shared one-hot vector, we must convey to E a valid share of one at position a. G thus sends the following value to E:

$$\left(\bigoplus_i L_i\right) \oplus \Delta$$

E XORs this value with the leaves she already holds and hence extracts $L_a \oplus \Delta$: a valid share of one.

Thus, the two parties compute $[\![\mathcal{H}(a)]\!]$ via $2(n-1)+1$ ciphertexts.

4.2 Garbled One-Hot Outer Product

We now generalize the above approach to compute $[\![\mathcal{H}(a) \otimes b]\!]$.

Let us back up to the point where the two parties each hold each L_i except that E does not hold L_a. For each j, the parties hold a garbled sharing $[\![b_j]\!]$. Let B_j (resp. $B_j \oplus b_j\Delta$) be G's (resp. E's) share.

For each $j \in [m]$ the parties act as follows. Both parties apply a PRG to each of their leaf seeds L_i and hence obtain strings $X_{i,j}$. Now, G sends to E the following value:

$$\left(\bigoplus_i X_{i,j}\right) \oplus B_j$$

E XORs this with her $2^n - 1$ values $X_{i \neq a,j}$ and with her share of b_j:

$$\left(\bigoplus_{i \neq a} X_{i,j}\right) \oplus \left(\left(\bigoplus_i X_{i,j}\right) \oplus B_j\right) \oplus (B_j \oplus b_j\Delta) = X_{a,j} \oplus b_j\Delta$$

In other words, at index a, E receives a share of b_j.

Thus, the parties now hold a sharing of a $2^n \times m$ matrix x where each row is all zeros except row a: row a holds the vector b. We have constructed $[\![\mathcal{H}(a) \otimes b]\!]$.

The full construction, formalized in Figure 3, requires G send to E $2(n-1)+m$ ciphertexts.

4.3 Applying the One-Hot Encoding

We now give some examples of how the one-hot outer product can be used. We greatly expand on this topic in Section 7.

Recall that garbled shares support linear maps (Lemma 3.5) and that for *any* function f the following equality holds:

$$\mathcal{T}(f)^\mathsf{T} \cdot \mathcal{H}(a) = f(a) \qquad \text{Lemma 3.3}$$

Zero Knowledge. We briefly mention that our one-hot outer product implies improvement for GC-based Zero Knowledge [JKO13, HK20b]; we emphasize that our focus is 2PC, not ZK.

In ZK, E knows *each* circuit wire value, so our requirement that E knows the argument to $\mathcal{H}(\cdot)$ is met automatically. Thus, in GC-ZK we can compute *any* function using only $2n-1$ ciphertexts by computing $\mathcal{T}(f)^\mathsf{T} \cdot [\![\mathcal{H}(a)]\!]$. However, we must keep the domain of f small, since the parties construct a tree with 2^n leaves.

If f requires a large circuit, then this truth-table based approach can improve over the circuit. For example, if the ZK proof invokes SHA256 on a small domain n-bit input, we need only $2n-1$ ciphertexts. The hand tuned SHA256 circuit, on the other hand, has a staggering 22573 AND gates [AAL+]. Other ZK protocols, e.g. [WYKW21], can similarly use truth tables by brute force constructing a one-hot encoding (at the cost of $O(2^n)$ AND gates). However, as the size of the input grows, our technique becomes more efficient. For tables with more than 9 input bits, our GC-based one-hot encoding will improve over other protocols.

2PC. We now consider 2PC applications where both parties have input and neither party knows any intermediate wire value.

Since our one-hot outer product primitive requires E to know the argument a, we must *reveal* a to E in cleartext. Of course, we cannot arbitrarily reveal cleartext values to E: this would not be secure. Instead, we are careful to only reveal values that have a mask applied such that the cleartext value remains protected.

We illustrate this idea by example. Let $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ be two bitstrings. Moreover, let n, m be small. (Formally, let n, m be at most logarithmic in the overall circuit input size. This restriction avoids exponential-time computation due to the one-hot technique.)

Suppose the parties hold two sharings $[\![a]\!]$ and $[\![b]\!]$ and wish to compute the (non-one-hot) outer product $[\![a \otimes b]\!]$. Note that outer products are useful since they can be leveraged to compute matrix products, integer products, and more (see Section 7).

First, G chooses two uniform masks $\alpha \in \{0,1\}^n$ and $\beta \in \{0,1\}^m$. The parties compute $[\![a \oplus \alpha]\!]$ and $[\![b \oplus \beta]\!]$ inside GC. Now, it is safe to reveal the values $a \oplus \alpha$ and $b \oplus \beta$ to E in cleartext. These values are revealed by G sending his color bits to E[3].

From here, the parties use the following straightforward lemma:

Lemma 4.1. *Let $x \in \{0,1\}^n, y \in \{0,1\}^m$ be two bitstrings and let* id : $\{0,1\}^n \rightarrow \{0,1\}^n$ *denote the identity function:*

$$\mathcal{T}(\text{id})^\mathsf{T} \cdot (\mathcal{H}(x) \otimes y) = x \otimes y$$

[3] Alternatively and more directly, G can *define* α (resp. β) to be his color bits of a (resp. b). This avoids sending small cleartext values to E and is similar to the method used in [ZRE15]. Here, we introduce the idea that G can send color bits of a masked value to E because this sending generalizes to non-XOR masks.

Proof.

$$\begin{aligned}
&\mathcal{T}(\text{id})^\mathsf{T} \cdot (\mathcal{H}(x) \otimes y) && \\
&= \mathcal{T}(\text{id})^\mathsf{T} \cdot (\mathcal{H}(x) \cdot y^\mathsf{T}) && \text{Definition } \otimes \\
&= (\mathcal{T}(\text{id})^\mathsf{T} \cdot \mathcal{H}(x)) \cdot y^\mathsf{T} && \text{Associativity} \\
&= \text{id}(x) \cdot y^\mathsf{T} && \text{Lemma 3.3} \\
&= x \cdot y^\mathsf{T} && \text{Definition id} \\
&= x \otimes y && \text{Definition } \otimes
\end{aligned}$$

□

In particular, the parties compute the following two values:

$$\mathcal{T}(\text{id})^\mathsf{T} \cdot [\![\mathcal{H}(a \oplus \alpha) \otimes b]\!] = [\![(a \oplus \alpha) \otimes b]\!]$$
$$\mathcal{T}(\text{id})^\mathsf{T} \cdot [\![\mathcal{H}(b \oplus \beta) \otimes \alpha]\!] = [\![(b \oplus \beta) \otimes \alpha]\!]$$

Finally, the parties compute the following:

$$\begin{aligned}
&[\![(a \oplus \alpha) \otimes b]\!] \oplus [\![(b \oplus \beta) \otimes \alpha]\!]^\mathsf{T} \oplus [\![\alpha \otimes \beta]\!] \\
&= [\![a \otimes b]\!] \oplus [\![\alpha \otimes b]\!] \oplus [\![b \otimes \alpha]\!]^\mathsf{T} \oplus [\![\beta \otimes \alpha]\!]^\mathsf{T} \oplus [\![\alpha \otimes \beta]\!] \\
&= [\![a \otimes b]\!] \oplus [\![\alpha \otimes b]\!] \oplus [\![\alpha \otimes b]\!] \oplus [\![\alpha \otimes \beta]\!] \oplus [\![\alpha \otimes \beta]\!] \\
&= [\![a \otimes b]\!]
\end{aligned}$$

(G knows $\alpha \otimes \beta$, so he can inject this value as a GC constant.)

Thus, E and G can compute the outer product $[\![a \otimes b]\!]$ using only two one-hot outer products. In total, G sends to E $3(n + m) - 4$ ciphertexts. This is a significant improvement compared to computing the outer product via AND gates: the AND-gate method consumes $2nm$ ciphertexts.

As an interesting aside, the above technique is a strict generalization of the [ZRE15] half-gates technique. Namely, if we consider length one inputs a and b, the above technique computes Boolean AND using only two ciphertexts. Moreover, the numbers of per-party calls to H match the half-gates technique.

While we have shown here only how to compute an outer product, our technique improves other functions as well (see Section 7). We highlight the key ideas common to such constructions:

1. Apply a mask to an internal circuit value such that it is safe to reveal the masked value to E.
2. Use the revealed value as input to a one-hot outer product.
3. Apply a function, via truth table, to this outer product matrix.
4. Use simple algebra to remove the introduced masks and obtain the desired output sharing. The parties can use the output in further GC evaluation.

4.4 A Framework for One-Hot Techniques

We found a number of interesting functions that can be efficiently implemented using the one-hot outer product (see Section 7). We certainly *did not* find *all* such functions. Thus, part of our contribution is a simple framework for designing new such constructions, which can then be directly used without building a new garbling scheme from the ground up.

Section 5 motivates and explains this framework in detail. In brief, notice our above high-level strategy involves revealing cleartext values to E. Our framework provides a simple infrastructure that prevents insecure leakage by packaging sensitive values into modules and ensuring these values cannot leave the module.

Our framework is a tool for designing modules that implement useful functions inside GC. Modules are built from a small set of primitives provided by the framework. These primitives allow the designer to specify what to compute, how to sample auxiliary randomness, and what to reveal to E. Crucially, the module designer *will not* directly manipulate garbled labels, material, and other garbling scheme artifacts – all such handling is done through the framework's primitives. In particular, this means that the module designer need not prove her GC instantiation secure: module security follows from our framework's security theorems.

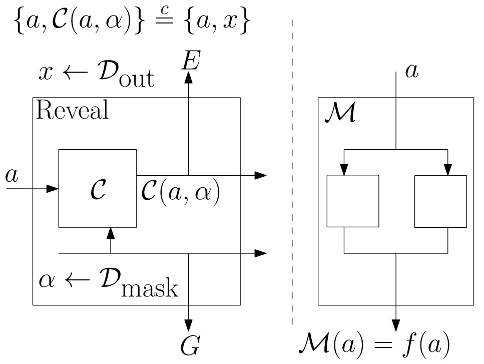
$\{a, \mathcal{C}(a,\alpha)\} \overset{c}{=} \{a, x\}$
$x \leftarrow \mathcal{D}_{\text{out}}$
E
Reveal
a
$\mathcal{C}$
$\mathcal{C}(a,\alpha)$
$\alpha \leftarrow \mathcal{D}_{\text{mask}}$
G
a
$\mathcal{M}$
$\mathcal{M}(a) = f(a)$

Figure 2: Left: Reveal gates safely reveal values to E. At runtime, G samples a mask α from the designer-specified distribution $\mathcal{D}_{\text{mask}}$. This mask, and the input, are fed into the designer-specified function C. For security, the output of C must be indistinguishable from a value sampled from some output distribution $\mathcal{D}_{\text{out}}$, even in the context of the input a (Requirement 2, Section 5.3). The masked value $C(a,\alpha)$ is revealed in cleartext to E and α is given to G. The masked value $C(a,\alpha)$ and the mask α are output as GC shares. Right: A module $\mathcal{M}$ implements a specific function and encapsulates any internal randomness that can emerge from Reveal gates. $\mathcal{M}$ must satisfy Requirement 1 (Section 5.3).

5 OUR GARBLING FRAMEWORK

In this section we formalize our approach. Typically, GC approaches consider simple gates, e.g. XOR/AND or ADD/MUL; the resulting GC framework is simple to prove and use. Ours is more complex.

At the heart of this complexity lies our highly nonstandard one-hot outer product gate. In particular, the gate is nonstandard because it requires that E know one of its inputs. Thus, to use the gate effectively, the GC must reveal certain values to E.

One direction we could take, but which we do not take, would be to expose one-hot gates to the circuit designer and to allow her to manage (e.g., via masking) the information release associated with its efficient use in GC. This would not be ideal, since each new top level circuit would require a new proof of security.

Instead, we do *not* allow our one-hot primitive to be used by top level circuits. Rather, these gates must be packaged into self-contained *modules*. Each module can use our primitive to efficiently implement a specific function. As a module might internally reveal values to E, it must satisfy certain *simple* security properties. Once these properties are proved, the module may be used by a top level circuit as if it were a standard gate.

INPUT: E inputs a. Parties together input shared bitstrings $[\![a]\!]$ and $[\![b]\!]$ where $a \in \{0,1\}^n, b \in \{0,1\}^m$.
OUTPUT: Parties output a shared matrix $[\![\mathcal{H}(a) \otimes b]\!]$.
PROCEDURE:

- Let A_i represent G's share of each bit a_i; hence E holds $A_i \oplus a_i\Delta$.
- Our first goal is to deliver to E $2^n - 1$ out of 2^n pseudorandom seeds where the ath seed is missing:
- E and G consider a full binary tree with 2^n leaves. Let $N_{i,j}$ be the jth node on level i and let the root reside on level -1.
- E and G label nodes from level 1 down with jointly agreed nonces $\text{nonce}_{i,j}$.
- G labels each node (except the root) with a κ-bit string $S_{i,j}$:
  - G labels $N_{0,0}$ by letting $S_{0,0} \triangleq A_0 \oplus \Delta$ and $N_{0,1}$ by letting $S_{0,1} \triangleq A_0$.
  - Consider $N_{i,j}$ with parent $N_{i-1,\lfloor j/2 \rfloor}$. G sets $S_{i,j} = H(S_{i-1,\lfloor j/2 \rfloor}, \text{nonce}_{i,j})$.
- For each level $i > 0$, G XORs all odd and all even labels:

$$\text{Even} \triangleq \bigoplus_{j=0}^{2^i-1} S_{i,2j} \qquad \text{Odd} \triangleq \bigoplus_{j=0}^{2^i-1} S_{i,2j+1}$$

  For each level $i > 0$, the parties agree on two nonces $\text{nonce}_{i,\text{even}}$ and $\text{nonce}_{i,\text{odd}}$. G sends to E the following two values:

$$H(A_i \oplus \Delta, \text{nonce}_{i,\text{even}}) \oplus \text{Even} \qquad H(A_i, \text{nonce}_{i,\text{odd}}) \oplus \text{Odd}$$

- E reconstructs each label $S_{i,j}$ except the labels along the path to leaf a:
  - E labels $N_{0,1}$ with A_0 if $a_0 = 0$; otherwise she labels $N_{0,0}$ with $A_0 \oplus \Delta$ (recall, her share is $A_0 \oplus a_0\Delta$).
  - Consider each level $i > 0$. There are two sibling nodes on level i that do not have a labeled parent. Consider each of the other $2^{i+1} - 2$ nodes $N_{i,j}$ with parent $N_{i-1,\lfloor j/2 \rfloor}$. E computes $H(S_{i-1,\lfloor j/2 \rfloor}, \text{nonce}_{i,j}) = S_{i,j}$.
  - For the other nodes on level i, E decrypts the XOR sum Even if a_i is odd or Odd if a_i is even; E XORs this value with her $2^i - 1$ even (resp. odd) labels and hence extracts the remaining even (resp. odd) label.
- G now holds 2^n strings $S_{n-1,j}$; E also holds each string except $S_{n-1,a}$. Rename these leaf strings $L_i \triangleq S_{n-1,i}$.
- For each bit b_j of b:
  - Let B_j be G's share of $[\![b_j]\!]$. Hence, E holds $B_j \oplus b_j\Delta$.
  - E and G agree on 2^n fresh nonces nonce_i.
  - For each leaf i, G sets $X_{i,j} \triangleq H(L_i, \text{nonce}_i)$. G sends to E:

$$\left(\bigoplus_i X_{i,j}\right) \oplus B_j$$

  - For each leaf $i \neq a$, E computes $X_{i,j} = H(L_i, \text{nonce}_i)$. E then computes:

$$\left(\bigoplus_{i \neq a} X_{i,j}\right) \oplus \left(\left(\bigoplus_i X_{i,j}\right) \oplus B_j\right) \oplus (B_j \oplus b_j\Delta) = X_{a,j} \oplus b_j\Delta$$

- Thus, for each column j of X, E and G hold 2^n values equal everywhere (i.e., each is a garbled share of zero) except at index a, where the parties hold an XOR share of $b_j\Delta$: the computation outputs a shared one-hot outer product.
- G outputs his matrix share X; E outputs her matrix share $X \oplus (\mathcal{H}(a) \otimes b)\Delta$

Figure 3: Our central primitive computes the outer product of two values: (1) a one-hot encoded value $\mathcal{H}(a)$ where E knows a and (2) a value b. For inputs $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$, G sends to E $2(n-1)+m$ ciphertexts. Recall from Section 3.3 that H is a circular correlation robust hash function.

In Section 7, we enumerate a number of useful modules, but we are confident that we have not found them all. Thus, we provide a *framework* for building and using modules: we specify the module requirements, and prove that, if met, the module can be used as a regular gate in GC. Thus GCs can be arbitrarily constructed from secure modules, without the need for additional proofs. New modules require proofs; the circuits that use them do not.

5.1 Reveal gates

Our framework introduces a *Reveal* meta-gate (see Figure 2). Reveal gates are our framework's method for revealing cleartext values to E. The GC may reveal a value to E so long as that value is indistinguishable from a value drawn from a fixed distribution (more formally, the input and output are *together* indistinguishable from the input and the sampled value). To achieve this indistinguishability, we allow the module designer to specify an arbitrary function that can apply a mask to the Reveal gate input value. The Reveal gate samples the mask (which is revealed to G) from a designer-specified distribution. In practice, this is achieved by G locally sampling the mask and programming it into the gate.

The Reveal gate produces as (garbled) output both the mask and the masked value; crucially, it also reveals in cleartext the masked value to E and the mask to G.[4] Because the masked value is indistinguishable from one drawn from a fixed distribution, our security proofs can simulate E's view[5].

[4] Values are revealed to E via color bits; as noted above, G selects the mask himself.

[5] In this work, we only use Reveal gates that each produce a distribution that is *identical* to a fixed distribution, not merely indistinguishable. We allow indistinguishability because it is more flexible and because it is easily proved secure.

OneHot.ev$(C, \mathbf{x})$ takes as input a circuit C and a string x. It outputs a string y. Each gate in a module is handled as follows:

- For each XOR gate $w_2 := w_0 \oplus w_1$ and each one-hot gate $w_2 := \mathcal{H}(w_0) \otimes w_1$, ev applies the appropriate function and stores the result in w_2. For each constant gate $w_0 := \text{Constant}(c)$, ev stores c in w_0.
- For each Module $w_1 := \text{Module}[\mathcal{M}](w_0)$, OneHot.ev recursively evaluates $\mathcal{M}$ on input w_0 and stores the result in w_1.
- For each Reveal gate $w_1^{\to E}, w_2^{\to G} := \text{Reveal}[C, \mathcal{D}_{\text{mask}}](w_0)$, OneHot.ev (1) samples a mask $\alpha \leftarrow \mathcal{D}_{\text{mask}}$, (2) recursively evaluates C with input (w_0, α), (3) stores the result on wire w_1, and (4) stores α on wire w_2. Each Color gate $w_1, w_2 := \text{Color}(w_0)$ is handled in the same manner as each Reveal gate except that α is drawn uniformly.

OneHot.Gb$(1^\kappa, C)$ takes as input a circuit C. It outputs an input encoding string e, an output decoding string d, and circuit material M. OneHot is a *projective* garbling scheme [BHR12], so e and d are standard. When OneHot.Gb is first called, it uniformly draws the global XOR offset $\Delta \leftarrow \{0,1\}^\kappa$ and sets Δ's least significant bit to one. To generate M, OneHot.Gb maintains a garbled share on each circuit wire. Each gate in a module is handled as follows:

- For each XOR gate $w_2 := w_0 \oplus w_1$, OneHot.Gb generates the output sharing by XORing the two input sharings (see Lemma 3.5).
- For each one-hot gate $w_2 := \mathcal{H}(w_0) \otimes w_1$, OneHot.Gb runs G's steps as described in Figure 3. When Figure 3 indicates G should send a message to E, OneHot.Gb appends the message to M.
- For each constant gate $w_0 := \text{Constant}(c)$, OneHot.Gb sets w_0 to $c\Delta$.
- For each Module, OneHot.Gb recursively garbles.
- For each Reveal gate $w_1^{\to E}, w_2^{\to G} := \text{Reveal}[C, \mathcal{D}_{\text{mask}}](w_0)$, Gb samples a mask $\alpha \leftarrow \mathcal{D}_{\text{mask}}$ and sets the share w_2 to $\alpha\Delta$. The procedure recursively garbles the subcircuit C with appropriate input shares and stores the output shares in w_1. Finally, the procedure evaluates Color(w_1) and attaches the result to M: informally, this reveals the masked output to E.
- For each Color gate $w_1, w_2 := \text{Color}(w_0)$, Gb computes Color$(w_0)$ and stores Color$(w_0)\Delta$ in w_2. It then stores $w_2 \oplus w_0$ in w_1.

OneHot.Ev(C, M, X) takes as input a circuit C, material M, and encoded input X. It outputs encoded output Y. OneHot.Ev maintains E's garbled share on circuit wires, propagating them through each gate in a module as follows:

- For each XOR gate $w_2 := w_0 \oplus w_1$, OneHot.Ev generates the output share by XORing the two input shares (see Lemma 3.5).
- For each one-hot gate $w_2 := \mathcal{H}(w_0) \otimes w_1$, OneHot.Ev runs E's steps as described in Figure 3. Note, w_0 must have been revealed for this call to be legal. When Figure 3 indicates G should send a message to E, OneHot.Ev parses the message from M.
- For each Constant gate $w_0 := \text{Constant}(c)$, OneHot.Ev sets w_0 to zero. Note, E need not know c.
- For each Module, OneHot.Ev recursively evaluates.
- For each Reveal gate $w_1^{\to E}, w_2^{\to G} := \text{Reveal}[C, \mathcal{D}_{\text{mask}}](w_0)$, Ev sets w_2 to zero. The procedure recursively evaluates the subcircuit C with appropriate input shares and stores the output shares in w_1. Finally, the procedure parses G's color bits col from M (see Gb above), computes Color$(w_1) \oplus$ col, and as a side-effect outputs this value: i.e, the value w_1 is revealed.
- For each Color gate $w_1, w_2 := \text{Color}(w_0)$, Ev computes Color$(w_0)$ and outputs this value to E. Color sets w_1 to w_0 and w_2 to zero.

OneHot.En(e, x) takes as input an encoding string e and cleartext input x; it outputs encoded input X. En maps each bit x_i to $X_i \oplus x_i\Delta$ for uniform X_i.

OneHot.De(d, Y) takes as arguments a decoding string d and an encoded output Y; it outputs a cleartext output y. We ensure that for each output bit y_i, d holds the following two strings:

$$H(Y_i, \text{nonce}) \qquad H(Y_i \oplus \Delta, \text{nonce})$$

where nonce is a fresh nonce. De hashes Y and outputs 0 or 1 depending on which above string matches. If neither matches, De aborts.

Figure 4: Our garbling scheme algorithms. We describe the handling of gates $\mathcal{G}$ (Equation (2)) inside of modules. Circuit/module handling is achieved by the repeated handling of gates. Note that our scheme does not directly provide XOR/AND gates to top-level circuits. However, these standard gates can be formalized as modules in our framework (see Section 5.4). Recall, in our notation wires hold matrices of bits.

We do not wish to restrict masking methods (in this work we mask via XORing, adding, and multiplying). Reveal gates can implement arbitrary masking by way of the circuit C (see Figure 2).

5.1.1 *Color Gates and Connection to [ZRE15].*

The half-gates technique views the color (see Section 3.2.2) of a GC label as a masked cleartext value, where the mask is known to G. They use this observation to help implement efficient AND gates.

Reveal gates can be viewed as a generalization of this simple masking: we allow *arbitrary* masks, and the chosen mask can be tailored to the application.

Reveal gates require G to send bits to E to reveal the output. Color bits do not require extra sending from G to E: the revealed value is implicit. We view color-based masking as a special case; for completeness, we include a special Color gate. At the interface, Color gates are the same as Reveal gates, except that they do not need a designer-specified distribution $\mathcal{D}_{\text{mask}}$ or circuit C. Color gates can be viewed as a specific instantiation of a Reveal gate.

Formally, a Color gate takes as input a matrix $[\![a]\!]$. Let α be the color of G's share: $\alpha = \text{Color}(A)$. The gate outputs (1) $[\![a \oplus \alpha]\!]$ and (2) $[\![\alpha]\!]$. The gate "reveals" α to G and $a \oplus \alpha$ to E. Of course, the parties already knew these values, so no communication is required – the Color gate is merely a formalism that allows modules to syntactically manipulate colors.

5.2 Modules

Reveal gates and Color gates do not *encapsulate* sensitive data that might be misused.

As an example, suppose the module designer specifies a Reveal gate that applies a uniform XOR mask α to a bit a: thus E learns $a \oplus \alpha$. Now suppose the designer inadvertently specifies that α is an output of the overall circuit. Because $a \oplus \alpha$ was revealed to E, this leaks a to E!

To achieve a clean modular GC framework, we must prevent sensitive values (i.e. values that depend on random masks) from escaping the context where they are used. Thus, we introduce the concept of *Modules*. A Module is a subroutine that computes a specific function of its input and that encapsulates internal data.

A Module is parameterized over a module-designer-provided circuit. It simply passes its input to this circuit and then propagates the output. So far, this is trivial. However, we require that the module designer provide a *proof of correctness* demonstrating that the internal circuit implements a deterministic function of its argument. This will, in particular, guarantee that the output is independent of internal random masks; thus, masks cannot escape the module.

Top level circuits are only allowed to manipulate modules. I.e., our one-hot outer product primitive, Reveal gates, and Color gates are *syntactically prohibited* outside of modules. This restriction means (we prove this) that top level circuits may use modules without any extra proofs, and so are suitable for end users.

5.3 Formal Syntax

We now formalize our syntax. Specifically, we formalize the space of circuits C, the space of modules $\mathcal{M}$, and the space of gates allowed in modules $\mathcal{G}$. Because we wish to allow modules to use other, simpler modules (i.e., one module designer should be allowed to use the work of another) our syntax is inductively defined.

As stated above, top level circuits are only allowed to manipulate modules. Formally, a circuit C is an ordered list of *modules* with specified input and output wires. Modules can manipulate lower level gates. A module $\mathcal{M}$ is a list of *gates* with accompanying input and output wires. Modules *do not* directly manipulate garbled shares, etc (Section 4.4).

We provide a grammar for the gates allowed inside of modules. Let each w_i denote a wire that holds a matrix of bits of arbitrary dimension. When a wire w_i's cleartext value is revealed to G (resp. E), we write $w_i^{\to G}$ (resp. $w_i^{\to E}$). When a wire is revealed, it remains a valid garbled sharing and can be used inside a module. Let $\mathcal{D}_{\text{mask}}$ refer to an arbitrary distribution over a finite set of values. Let c refer to a constant chosen by G:

$$\begin{aligned}\mathcal{G} \triangleq\ & w_2 := w_0 \oplus w_1 \\ \mid\ & w_2 := \mathcal{H}(w_0) \otimes w_1 \\ \mid\ & w_0 := \text{Constant}(c) \\ \mid\ & w_1^{\to E}, w_2^{\to G} := \text{Reveal}[C, \mathcal{D}_{\text{mask}}](w_0) \\ \mid\ & w_1^{\to E}, w_2^{\to G} := \text{Color}(w_0) \\ \mid\ & w_1 := \text{Module}[\mathcal{M}](w_0)\end{aligned} \tag{2}$$

That is, modules can use gates that (1) compute the XOR of two matrices (of equal dimension), (2) compute the one-hot outer product of two vectors, (3) output a constant chosen by G, (4) reveal a masked value to E (see Figure 2), (5) incorporate a share's color in the GC (see Section 5.1.1), and (6) recursively call another module. We refer to one-hot outer product gates simply as 'one-hot' gates.

We specify two requirements each module *must* satisfy.

Requirement 1 (Module correctness). *For module $\mathcal{M}$ computing function f, it must hold that for all inputs x:*

$$f(x) = \mathcal{M}(x)$$

Requirement 2 (Reveal indistinguishability). *For a Reveal gate $w_1^{\to E}, w_2^{\to G}$:= Reveal$[C, \mathcal{D}_{\text{mask}}](w_0)$, there must exist a distribution $\mathcal{D}_{\text{out}}$ such that for all inputs x on wire w_0 and for $r \leftarrow \mathcal{D}_{\text{out}}$ and $\alpha \leftarrow \mathcal{D}_{\text{mask}}$ the following indistinguishability holds:*

$$\{x, C(x, \alpha)\} \overset{c}{=} \{x, r\}$$

Note, each module may have more than one Reveal gate, so it may not be *a priori* clear that arbitrary Reveal gate interactions are secure. For instance, is it safe to feed the output of one Reveal gate as input to another? From Requirement 2, we can prove that everything revealed in a module can be simulated by a fixed distribution. We then (Theorem 5.5) prove that this is sufficient for security.

Lemma 5.1. *Let $\mathcal{M}$ be a module and let y be the tuple of all values revealed to E in $\mathcal{M}$ due to Reveal gates and Color gates (as formally specified by OneHot.Ev). There exists a distribution $\mathcal{D}_{rev}$ such that:*

$$\{y\} \overset{c}{=} \{r\} \qquad\qquad \textit{where } r \leftarrow \mathcal{D}_{rev}$$

Specifically, $\mathcal{D}_{\text{rev}}$ is the distribution that samples from each Reveal gate distribution $\mathcal{D}_{\text{out}}$ (and samples a uniform distribution in the case of Color gates) and concatenates the samples. Due to lack of space, we prove Lemma 5.1 in Appendix A.

5.4 Standard Boolean Gates in Our Framework

Neither XOR nor AND are *by default* available to top level circuits. However, these functions can be expressed as modules, and thus traditional Boolean circuits are compatible with our framework:

XOR is easily handled by building a module with a single XOR gate that XORs the two bits of its input and outputs the result.

AND is conspicuously missing from $\mathcal{G}$. We do not need a separate AND gate primitive, because we can express AND as a module. Moreover, the resulting module is *functionally identical* to the state-of-the-art half-gates technique [ZRE15]. Namely, our approach uses the same number of calls to H for each party and transfers the same number of ciphertexts (i.e., two) from G to E.

Let a, b be bits, and view them as one element vectors. Let α, β denote the color of a, b respectively. Note the following equality:

$$\begin{aligned}&(\mathcal{T}(\text{id}) \cdot \mathcal{H}(a \oplus \alpha) \otimes b) \oplus (\mathcal{T}(\text{id}) \cdot \mathcal{H}(b \oplus \beta) \otimes \alpha) \oplus \alpha\beta \\ &= ((a \oplus \alpha) \otimes b) \oplus ((b \oplus \beta) \otimes \alpha) \oplus \alpha\beta \\ &= ((a \oplus \alpha)b) \oplus ((b \oplus \beta)\alpha) \oplus \alpha\beta \\ &= ab\end{aligned}$$

Thus, we can compute ab via two Color gates (to compute and reveal $a \oplus \alpha$ and $b \oplus \beta$), two one-hot gates, one Constant gate (for $\alpha\beta$), and XOR gates (to compute $\mathcal{T}(id)\cdot$). Each of these sub-components is communication free except for the one-hot gates. A one-hot gate uses $2(n-1)+m$ ciphertexts; here, in both cases $n=m=1$, so the module costs a total of two ciphertexts.

Thus, [ZRE15] half-gates can be hosted in our framework.

5.5 The OneHot Garbling Scheme

Now that we have established syntax, we prove the framework secure. We formalize our framework as a *garbling scheme* [BHR12].

A garbling scheme is a five-tuple of algorithms:

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

These five algorithms specify the actions taken by G and E when executing the protocol. Informally, (1) En describes how cleartext inputs are encoded as garbled shares, (2) Gb describes how G constructs the garbled circuit, (3) Ev describes how E uses input shares and the garbled circuit to compute output shares, (4) De describes how output shares are decoded to cleartext outputs, and (5) ev provides a cleartext specification of the circuit semantics. Loosely speaking, En, Gb, Ev, and De should together perform the same task as ev while preventing E from learning G's inputs.

Construction 1 (OneHot Garbling Scheme). *OneHot is the tuple of algorithms defined in Figure 4 by reference to Figure 3.*

Our scheme is a straightforward formalization of the high level intuition given in Section 4.

OneHot satisfies the [BHR12] definitions of *correctness*, *obliviousness*, *privacy*, and *authenticity*. We include definitions and explanations of each of these properties. **Full formal proof of each theorem is presented in Appendix A due to a lack of space.**

Definition 5.2 (Correctness). A garbling scheme is *correct* if for all circuits C and all inputs x:

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

where $(M, e, d) \leftarrow \text{Gb}(1^\kappa, C)$.

Correctness requires the scheme to realize the semantics specified by ev. That is, the implementation matches the specification.

Theorem 5.3. *OneHot is correct.*

Correctness is mostly trivial, save the correctness of one-hot gates. One-hot gate correctness can be inferred from discussion in Section 4. See Appendix A for a full proof.

Definition 5.4 (Obliviousness). A garbling scheme is *oblivious* if there exists a simulator $\mathcal{S}_{\text{obv}}$ such that for any circuit C and all inputs x, the following are indistinguishable:

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

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

Informally, obliviousness ensures that the material M and encoded input shares X reveal no information about the input x or about the output $\text{ev}(C, x)$.

Theorem 5.5. *If H is a circular correlation robust hash function, then OneHot is oblivious.*

In short, because of the properties of H we can simulate most values by uniform bits. For Reveal gates, we instead simulate values by sampling from each such gate's specified distribution; this is valid due to Requirement 2. See Appendix A for a full proof.

Definition 5.6 (Privacy). A garbling scheme is *private* if there exists a simulator $\mathcal{S}_{\text{prv}}$ such that for any circuit C and all inputs x, the following are computationally indistinguishable:

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

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

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

Theorem 5.7. *If H is a circular correlation robust hash function, then OneHot is private.*

The privacy simulator follows relatively trivially from the obliviousness simulator and from our choice of output decoding string d (Figure 4). See Appendix A for a full proof.

Definition 5.8 (Authenticity). A garbling scheme is *authentic* if for all circuits C, all inputs x, and all poly-time adversaries $\mathcal{A}$ the following probability is negligible in κ:

$$\Pr\left(Y' \neq \text{Ev}(C, M, X) \wedge \text{De}(d, Y') \neq \bot\right)$$

where $(M, e, d) = \text{Gb}(1^\kappa, C)$, $X = \text{En}(e, x)$, and $Y' = \mathcal{A}(C, M, X)$.

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

Theorem 5.9. *If H is a circular correlation robust hash function, then OneHot is authentic.*

Authenticity is nontrivial only for one-hot gates. One-hot gates can be shown authentic due to the properties of H. See Appendix A for a full proof.

5.5.1 Compatibility with Stacked Garbling.

As mentioned in Section 2, stacked garbling (SGC) is a state-of-the-art GC improvement for conditional branching [HK20a, HK21]. SGC is parameterized over an underlying garbling scheme which it leverages to handle each conditional branch. OneHot can, in a slightly limited sense, be used as this underlying scheme.

SGC requires that the underlying scheme produces garbled material M and inputs shares X that are indistinguishable from uniform strings. Our scheme satisfies this, with the notable exception of Reveal gates. OneHot can be 'stacked' so long as all Reveal gates use uniform binary strings as their output distribution $\mathcal{D}_{\text{out}}$.

In Appendix A, we prove that, under this condition, OneHot is *strongly stackable* [HK21] and can be the SGC underlying scheme.

6 EXPERIMENTAL SETUP

In the following section, we give experimental findings of the performance of our technique as compared to standard Boolean circuits. We record details of our experimental setup here.

Implementation Details. We implemented our technique and benchmarks in ∼ 2000 lines of C++. Our implementation uses our garbling scheme to instantiate a semi-honest 2PC protocol. Garbled shares are 128 bits long. Hence our security parameter $\kappa = 127$; the 128th bit is reserved for share color.

We compare our implementation against half-gates [ZRE15]. We refer to half-gates based implementations of our experiments simply as 'standard'. We do not compare in detail to the concurrent

Input: Parties input shared bitstrings $[\![a]\!]$, $[\![b]\!]$ where $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$.
Output: Parties output a shared matrix $[\![a \otimes b]\!]$.
Procedure:

- Parties reveal to E $a \oplus \alpha$ and $b \oplus \beta$ for uniform $\alpha \in \{0,1\}^n, \beta \in \{0,1\}^m$ via Color gates. Notably, the Color gates output $[\![a \oplus \alpha]\!]$, $[\![b \oplus \beta]\!]$, and $[\![\alpha]\!]$.
- Parties compute $[\![\mathcal{H}(a \oplus \alpha) \otimes b]\!]$ via a one-hot gate.
- Parties compute $[\![\mathcal{H}(b \oplus \beta) \otimes \alpha]\!]$ via a one-hot gate.
- Parties compute the following two outer products:

$$\mathcal{T}(\text{id})^\intercal \cdot [\![\mathcal{H}(a \oplus \alpha) \otimes b]\!] = [\![(a \oplus \alpha) \otimes b]\!] \qquad \text{Lemma 4.1}$$
$$\mathcal{T}(\text{id})^\intercal \cdot [\![\mathcal{H}(b \oplus \beta) \otimes \alpha]\!] = [\![(b \oplus \beta) \otimes \alpha]\!] \qquad \text{Lemma 4.1}$$

- G locally computes $\alpha \otimes \beta$. He injects $[\![\alpha \otimes \beta]\!]$ as a constant.
- Parties compute and output:

$$[\![(a \oplus \alpha) \otimes b]\!] \oplus [\![(b \oplus \beta) \otimes \alpha]\!]^\intercal \oplus [\![\alpha \otimes \beta]\!] = [\![a \otimes b]\!]$$

See Section 4.3 for a correctness argument.

Figure 5: Efficient small domain outer product module. The module implements the function $a, b \mapsto a \otimes b$.

work [RR21]; moreover their technique has not yet been implemented. For many of our applications, our improvement will be slightly diminished given a fast [RR21] implementation. In particular, our work improves over [RR21] for all considered applications, except for AES S-Box.

Computation Setup. For each experiment, we ran both G and E on a single commodity laptop: a MacBook Pro with an Intel Quad-Core i7 2.3GHz processor and 16GB of RAM. The two parties run in parallel on separate processes on the same machine.

Communication Setup. G and E communicate over a simulated 100Mbps WAN. (For completeness we configure the network with 30ms latency, though this is largely irrelevant in our experiments which do not incur multiple rounds of interaction.)

In our experiments, we record bandwidth consumption and wall clock time. For each experiment, we build a top-level circuit that repeatedly uses the target module 1000 times; our presented measurements divide total communication/total wall clock time by 1000 to approximate the cost of a single module instance.

7 APPLICATIONS

In this section, we instantiate applications of our approach. Each application is formalized in our framework (see Section 5); when necessary, we implement a module.

We mention that all of the following modules, with the exception of our binary field inverse and our modular reduction, are compatible with stacked garbling.

7.1 Small Domain Binary Outer Products

Our first module follows naturally from our one-hot primitive. Let $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ be two bitstrings and let n, m be small (formally, at most logarithmic in the overall circuit input size). The module maps two input garbled sharings $[\![a]\!]$, $[\![b]\!]$ to the outer product $[\![a \otimes b]\!]$.

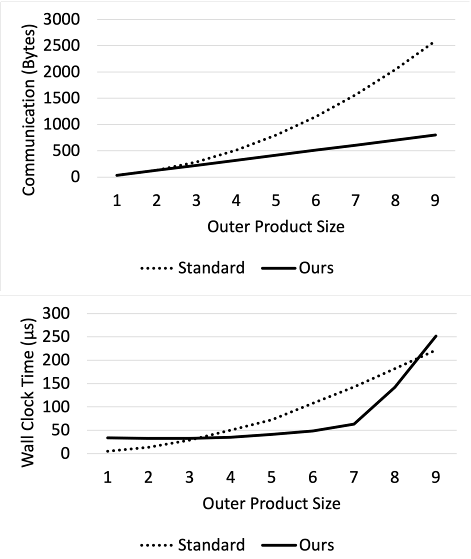

Figure 6: Communication consumption (top) and wall clock time (bottom) when computing the outer product of two n-bit vectors. We varied n from 1 to 9. The standard method computes the outer product using AND gates. Our technique's computation scales exponentially in the vector sizes, but is more efficient for vectors between lengths 4 and 8.

This module was explained in Section 4 and is formalized in Figure 5. Because we need XOR-based masks, we use Color gates.

The full construction consumes only $3(n+m)-4$ ciphertexts, a significant improvement from the $2nm$ ciphertexts needed to compute the outer product via AND gates.

We implemented our module and experimented with its performance. Figure 6 plots the results.

7.2 General Binary Outer Products

We have shown how to compute the outer product of two *short* vectors. We are, so far, limited to short vectors because of the exponential computation scaling of our one-hot technique.

It is interesting to compute the outer product of vectors of all sizes, not just short ones. Here, we give an efficient construction of general outer products.

In Section 7.1 we decomposed $a \otimes b$ into three summands:

$$(a \otimes b) = ((a \oplus \alpha) \otimes b) \oplus ((b \oplus \beta) \otimes \alpha)^\intercal \oplus (\alpha \otimes \beta)$$

The third term is known to G and is free. The other two terms must be computed inside the GC. Consider the term $(a \oplus \alpha) \otimes b$.

In Section 7.1 we insisted that this outer product be computed by a single one-hot gate. More generally, we can *tile together* multiple one-hot outer products. We ensure the tiles are small enough that computation remains polynomial in the input size.

Each tile computes the outer product of a k-bit chunk of $a \oplus \alpha$ with b, yielding a $k \times m$ submatrix of the full outer product $(a \oplus \alpha) \otimes b$. Vertically concatenating the $\lceil n/k \rceil$ submatrices yields the correct

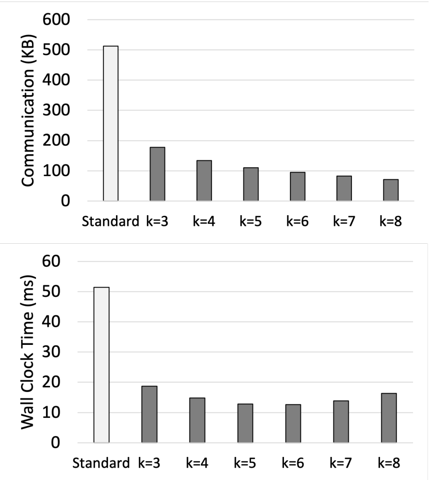

Figure 7: We used our implementation to compute the bitwise outer product of two 128 bit vectors. We instantiated our approach with various "chunking factors" k (see Section 7.2). Increasing k decreases communication but increases computation, due to the exponential computation scaling of our one-hot gate. The standard method computes outer products by simply ANDing pairs of values. At $k = 6$, we improve over standard by $6.2\times$ (communication) and $4.1\times$ (time).

result. We use the same idea to compute $(b \oplus \beta) \otimes \alpha$. We defer formal presentation of the module to Appendix B.

If the chosen chunk size k is logarithmic in the size of input, then the parties compute $a \otimes b$ in polynomial time. In terms of communication, the parties use $O(nm/k)$ garbled rows: a factor k improvement over the standard method. Formally, we improve outer product communication by a logarithmic factor; in practice we choose constants k that yield good performance.

Figure 7 plots the practical efficiency we obtained when implementing general outer products with different values of k. The results show that our approach significantly improves outer products over prior state-of-the art.

7.3 Binary Matrix Multiplication

It is well known that outer products can be used to efficiently multiply matrices. We implemented this approach – see Appendix B. For chunking factor $k = 6$, our approach improves 128×128 square matrix multiplication by 6.2× (communication) and 5× (time).

7.4 Integer Multiplication

Consider bit vectors $a, b \in \{0, 1\}^n$ that each represent n-bit numbers. The outer product of $a \otimes b$ can be used to help calculate the product $a \cdot b$. See Appendix B for further discussion.

We implemented 32-bit integer multiplication using our technique and the standard method (our standard circuit is inspired by [WMK16]). Best performance was achieved with "chunking factor" (see Section 7.2) $k = 6$:

	Standard	Ours	Improvement
Comm. (KB)	32.0	21.3	1.51×
Time (ms)	3.20	2.32	1.38×

As compared to outer products and matrix multiplication, our improvement here is less substantial: after the outer product is computed, our technique still must add together values in the standard manner. Still, we achieve improvement to an important primitive.

In the GC setting, the Karatsuba fast multiplication method improves over standard multiplication even for small 20-bit integers [HKS+10]. Karatsuba is a recursive divide-and-conquer algorithm. At the leaves of the recursion (i.e. for 19-bit numbers or less), it is best to use standard multiplication. We thus can use our improved standard multiplication method to accelerate Karatsuba-based multiplication.

7.5 Binary Field Multiplication

Consider an arbitrary binary field $GF(2^n)$. In such fields, multiplication can be understood as polynomial multiplication modulo an irreducible polynomial $p(x)$. By representing elements $a, b \in GF(2^n)$ as *vectors* of bits, we can easily compute the product of the two polynomials from the vector outer product. Once computed, the product can be reduced modulo $p(x)$ by a linear function [GKPP06]. Thus, our outer product construction improves binary field multiplication by the "chunking factor" k (see Section 7.2).

Because this multiplication only uses a black box outer product followed by XORs, we do not need to formalize a module.

We implemented both our approach and a standard circuit for $GF(2^8)$ (modulo $x^8 + x^4 + x^3 + x + 1$). We used the best available standard circuit for this field [BDP+20]. We ran our version with chunking factor $k = 4$ and $k = 8$. We list communication, wall clock time, and corresponding improvement over standard:

	Standard	$k = 4$		$k = 8$	
Comm. (Bytes)	1536	896	1.71×	704	2.18×
Time (μs)	146	80	1.82×	111	1.3×

Despite the fact efficient hand-tuned circuits are available, we improve communication consumption by more than 2×.

7.6 Binary Field Inverses and the AES S-Box

Our technique can compute binary field inverses using less communication than the state-of-the-art. Consider a field $GF(2^n)$ where n is small (formally, logarithmic in the circuit input size). Let $a \in GF(2^n)$ be a field element and suppose $a \neq 0$ (we handle this separately).

Our module follows from a technique given by [BIB89]. Namely, for non-zero input a, we first compute $a \cdot \alpha$ for uniform non-zero mask α. Then, we reveal $a \cdot \alpha$ to E. With this done, we use a one-hot gate to efficiently compute $(a \cdot \alpha)^{-1} \cdot \alpha = a^{-1}$. Due to a lack of space, we defer a full formalization of our inverse module to Appendix B.

S-Boxes. The AES S-Box, which is the only non-linear component of the AES block cipher, performs a single inversion in $GF(2^8)$; all other parts of the S-Box are linear. The state-of-the-art Boolean circuit S-Box uses 32 AND gates [BP10]. Thus, with the half-gates technique, this implementation consumes 64 ciphertexts.

Our full inverse gate consumes 58 ciphertexts: 22 to compute $[\![a \cdot \alpha]\!]$, 22 to then compute the inverse, and 14 to handle the case where $a = 0$. This improves communication by $\sim 10\%$.

We implemented the [BP10] S-Box and our one-hot version:

	Standard	Ours	Improvement
Comm. (Bytes)	1024	929	1.10×
Time (μs)	103.6	105.8	0.98×

On a WAN, our implementation is slightly slower than the standard S-Box. This can likely be improved by low-level code optimization.

It may be possible to further apply our technique to block ciphers, perhaps by codesigning with our new cost structure in mind. We leave such fine-grained approaches to future work.

7.7 Modular Reduction

Let $x \bmod y$ denote a *function* that computes the remainder of x divided by y. Suppose the parties hold a sharing $\llbracket a \rrbracket$ and wish to compute $\llbracket a \bmod \ell \rrbracket$ where ℓ is a public constant. Such computation is potentially useful, e.g. to compute in an arithmetic field $\mathbb{Z}_p$.

The Boolean circuit that computes $(\cdot) \bmod \ell$ is an expensive quadratic construction. One-hot gates can improve the cost.

Our module first subtracts a random mask α from a and then reveals $a - \alpha$ to E. It then splits a into small k-bit "chunks" and, for each chunk, efficiently computes $(\cdot) \bmod \ell$ using a one-hot gate. The reduced chunks can then be recombined and the mask stripped off using addition mod ℓ. (Addition modulo ℓ where both arguments are already less than ℓ is a special case and can be computed efficiently.) Crucially, the number of needed additions is proportional only to the number of chunks. Due to lack of space, we defer formal treatment of the module to Appendix B.

For our concrete experiment, we implemented modular reduction for 32-bit numbers using the prime modulus $p = 65521$ (the largest 16-bit prime). Our standard implementation conditionally subtracts $p \cdot 2^k$ for $k \in [16]$; thus 16 conditional subtractions are needed. Our optimized version uses chunking factor $k = 8$. The technique requires only 3 additions and 3 conditional subtractions and hence substantially improves performance:

	Standard	Ours	Improvement
Comm. (KB)	35.1	10.5	3.3×
Time (ms)	3.75	1.08	3.5×

7.8 Exponentiation

Suppose the parties hold a sharing $\llbracket a \rrbracket$ and wish to compute $\llbracket \ell^a \rrbracket$ where ℓ is a publicly agreed constant. For special cases of ℓ (e.g., $\ell = 2$), there are fast circuits that compute $\llbracket \ell^a \rrbracket$. However, for arbitrary ℓ we need to repeatedly multiply inside GC, which is expensive. We can use one-hot gates to greatly reduce the number of needed multiplications.

Our module first subtracts a uniform additive mask α from a and then reveals $a - \alpha$ to E. Then, the module splits $\llbracket a - \alpha \rrbracket$ into small k-bit "chunks" and, for each chunk c, computes $\llbracket \ell^c \rrbracket$ using a one-hot gate. These intermediate values can be combined and the mask stripped off using multiplication. We use our improved multiplication technique (Section 7.4) to further improve the module. We defer formal treatment of the module to Appendix B.

We implemented exponents for 32-bit numbers using a standard technique (which consumes 31 standard multiplications) and our technique (with chunking factor $k = 8$, which consumes only 4 improved multiplications):

	Standard	Ours	Improvement
Comm. (KB)	1024	87	11.8×
Time (ms)	101	10.6	9.52×

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

REFERENCES

[AAL+] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. 'bristol fashion' mpc circuits. https://homes.esat.kuleuven.be/~nsmart/MPC.

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 223–254. Springer, Heidelberg, August 2017.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BDP+20] Joan Boyar, Morris Dworkin, Rene Peralta, Meltem Turan, Cagdas Calik, and Luis Brandao. Circuit Minimization Work. http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html, 2020.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

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

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

[BIB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, April 2013.

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

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

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. *Experimental Algorithms Lecture Notes in Computer Science*, page 178–189, 2010.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[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.

[DKS+17] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS 2017*. The Internet Society, February / March 2017.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE

Computer Society Press, October 1984.

[GKPP06] Jorge Guajardo, Sandeep S. Kumar, Christof Paar, and Jan Pelzl. Efficient software-implementation of finite fields with applications to cryptography. In *Acta Applicandae Mathematica*, 2006.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, May 2020.

[GLMY16] Adam Groce, Alex Ledger, Alex J. Malozemoff, and Arkady Yerukhimovich. CompGC: Efficient offline/online semi-honest two-party computation. Cryptology ePrint Archive, Report 2016/458, 2016. https://eprint.iacr.org/2016/458.

[GLNP18] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. *Journal of Cryptology*, 31(3):798–844, July 2018.

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

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

[HK21] David Heath and Vladimir Kolesnikov. Logstack: Stacked garbling with o(b log b) computation. Cryptology ePrint Archive, Report 2021/531, 2021. https://eprint.iacr.org/2015/751.pdf.

[HKP20] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In *ASIACRYPT 2020, Part III*, LNCS, pages 3–30. Springer, Heidelberg, December 2020.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010*, pages 451–462. ACM Press, October 2010.

[IKM+13] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 600–620. Springer, Heidelberg, March 2013.

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

[KKK+15] Matthew Kelly, Alan Kaminsky, Michael Kurdziel, Marcin Łukowiak, and Stanisław Radziszowski. Customizable sponge-based authenticated encryption using 16-bit s-boxes. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*, pages 43–48, 2015.

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

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

[KNR+17] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 3–20. ACM Press, October / November 2017.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

[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.

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

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

[PSSY20] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. Cryptology ePrint Archive, Report 2020/1225, 2020. https://eprint.iacr.org/2020/1225.

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

[RWT+18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, April 2018.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1055–1072. ACM Press, November 2019.

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

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

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1607–1626. ACM Press, November 2020.

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

A FORMAL PROOFS

We prove Lemma 5.1.

Proof. By constructing $\mathcal{D}_{\text{rev}}$.

In this proof, we view Color gates as a strict special case of Reveal gates that use a uniform output distribution $\mathcal{D}_{\text{out}}$.

Recall that $\mathcal{D}_{\text{rev}}$ must simulate *all* values revealed by Reveal gates in a module $\mathcal{M}$. $\mathcal{D}_{\text{rev}}$ is the distribution that, when sampled, samples each Reveal gate's distribution $\mathcal{D}_{\text{out}}$ and concatenates the results.

We show this secure by a hybrid argument. Let the considered module $\mathcal{M}$ have n Reveal gates. Let hybrid h_0 be the real distribution of revealed values and let hybrid h_n be $\mathcal{D}_{\text{rev}}$. Each hybrid h_i is equal to h_{i-1} except that we replace the output of the ith Reveal gate by a sample from its distribution $\mathcal{D}_{\text{out}}$.

Assume the real distribution and $\mathcal{D}_{\text{rev}}$ are distinguishable, and hence for some i, h_{i-1} and h_i allow a distinguisher. Thus, the ith gate's simulated output must allow the distinguisher. Let $y^{\to E}, z^{\to G} :=$ $\text{Reveal}[C, \mathcal{D}_{\text{mask}}](x)$ denote the ith Reveal gate. Requirement 2 gives the following property for *all* inputs x:

$$\{x, C(x, \alpha)\} \overset{c}{=} \{x, r\} \qquad \text{where } r \leftarrow \mathcal{D}_{\text{out}}, \alpha \leftarrow \mathcal{D}_{\text{mask}}$$

This property implies that the output of the ith gate is (computationally) independent of its input. Hence the output of the ith gate *must* be computationally independent of all other Reveal gate outputs. Note, (1) $\{C(x, \alpha)\} \overset{c}{=} \{r\}$ holds trivially from the above property and (2) the presence of other simulated Reveal outputs cannot help a distinguisher because each such output is mutually independent with every other output (up to computational indistinguishability). Therefore, the existence of distinguisher between h_{i-1} and h_i contradicts Requirement 2, and hence our assumption does not hold.

As an informal aside, note that this lemma would *not* hold if we required only that the output of a Reveal gate is indistinguishable from $\mathcal{D}_{\text{out}}$: by requiring that indistinguishability hold even in the context of the Reveal gate input, we ensure that Reveal outputs are independent of one another (and hence cannot even *jointly* reveal private information). □

We prove Theorem 5.3: OneHot is correct.

Proof. By induction on a module $\mathcal{M}$. Our inductive hypothesis is that modules are correct. Once modules are proved correct, circuits are trivially correct, since a circuit is merely a list of modules.

To prove modules correct, we first argue that each individual gate type is in some sense correct. Note that we cannot prove gates *strictly* correct because both Reveal and Color are *non-deterministic*. However, we *can* show that in both our implementation (OneHot.Gb and OneHot.Ev) and our specification (OneHot.ev), gates produces outputs in the same *distribution*. This, combined with the module-designer proof of correctness (Requirement 1), suffices to show that the module as a whole is strictly correct.

We proceed by case analysis on gates.

XOR gates use the linearity of garbled shares (Lemma 3.5), and are trivially correct.

Constant gates are trivially correct: For constant c, G uses share $c\Delta$ and E uses share 0. This is a valid garbled sharing of c (Definition 3.4).

One-hot gates are our most complex construction. Figure 3 explains many details inline; we expand on details here.

We consider a one-hot gate with inputs a and b that computes $\mathcal{H}(a) \otimes b$. Recall that E is assumed to know a. The two parties begin by jointly expanding a GGM tree such that, in the end, G computes 2^n leaf nodes L_i and E computes each leaf node $L_{i \neq a}$.

We prove this initial expansion correct by induction on the levels of the tree; namely for each level i, G holds 2^{i+1} strings $S_{i,j}$ and E holds all such strings except the single string on the path to L_a. In the base case, G chooses $S_{0,0}$ to be $A_0 \oplus \Delta$ and $S_{0,1}$ to be A_0. Thus the invariant trivially holds.

Now consider arbitrary level i such that level $i-1$ is already populated. E trivially expands all strings on this level save two: one on the path to a (which she should not receive) and one *just off* the path to a. To correct for this, G sends to E two encryptions that allow her to obtain the XOR sum of all even strings/all odd strings depending on her share $A_i \oplus a_i\Delta$. From this sum, she XORs on all of her already expanded even (resp. odd) strings and recovers the missing even (resp. odd) string. From this, E recovers the string just off the path to L_a. Hence the inductive invariant holds.

Next, the parties expand their leaf strings into garbled sharings $[\![\mathcal{H}(a) \otimes b]\!]$. G generates all of his shares simply by applying H to each leaf m times. Thus G computes a $2^n \times m$ matrix. E similarly applies H to her leaves m times. Thus, she obtains the same matrix as G except that row a is missing.

Let X be G's matrix. Let B_j be G's share of bit b_j. For each column j of the matrix, G computes and sends to E the following value:

$$\left(\bigoplus_i X_{i,j}\right) \oplus B_j$$

Note that E holds her garbled share $B_j \oplus b_j\Delta$. Thus, she computes:

$$\left(\bigoplus_{i \neq a} X_{i,j}\right) \oplus \left(\left(\bigoplus_i X_{i,j}\right) \oplus B_j\right) \oplus \left(B_j \oplus b_j\Delta\right) = X_{a,j} \oplus b_j\Delta$$

Thus, in row a, the parties hold garbled shares of b. Altogether, the parties hold a sharing $[\![\mathcal{H}(a) \otimes b]\!]$.

One-hot gates are correct.

Recursive Module calls are correct by induction.

Reveal gates are correct by inspection. The specification and the implementation of Reveal gates match: both procedures sample values from $\mathcal{D}_{\text{mask}}$ and feed them as input to the internal masking circuit. The internal masking circuit is correct by induction.

Color gates are similarly correct: the specification draws a uniform value while the implementation uses the color bit, which is uniform. Note there is one tedious detail here: If we are being pedantic, calling a color gate on the same input more than once is problematic because the implementation uses the *same* uniform bit for each gate, whereas the specification draws a *fresh* mask for each gate. This is easily remedied by having the specification associate a uniform color with each wire. We elide this detail outside of this discussion because it is so minor.

We have now shown each gate type correct in the sense that the implementation and specification produce equal distributions. Now, the module that calls these gates is *strictly* correct, because the module designer provided a proof of correctness that demonstrates the module output is independent of any internal randomness (Requirement 1).

Since modules are correct, circuits are correct. OneHot is correct. □

We prove Theorem 5.5: If H is a circular correlation robust hash function, then OneHot is oblivious.

Proof. By construction of a simulator $\mathcal{S}_{\text{obv}}$. At a high level, all messages sent from G to E are simulated by uniform bits, except values leaked by Reveal gates which are instead simulated by sampling each such gate's specified output distribution (Requirement 2).

First, $\mathcal{S}_{\text{obv}}$ uniformly samples input shares X'. In isolation, these are trivially indistinguishable from the real shares X, because each share in X is drawn uniformly (with Δ conditionally added). These remain indistinguishable in the context of C and M.

We describe the simulator's gate-by-gate handling and argue that the resultant material (even in context of input shares) is indistinguishable. The simulator propagates the simulated input shares X' to simulated output shares Y' and builds up material M'. Our indistinguishability argument proceeds by induction on the structure of a module $\mathcal{M}$. The inductive hypothesis maintains that the simulated garbling of each submodule is indistinguishable from the real garbling.

XOR gates are handled simply: $\mathcal{S}_{\text{obv}}$ XORs the input shares. No change is made to the simulator's output.

Constants are also simple: $\mathcal{S}_{\text{obv}}$ sets the output share to zero.

One-hot gates are more involved.

Recall that a one-hot gate first proceeds level-by-level through a binary branching tree. For each level, Gb includes an encryption of all even nodes and of all odd nodes:

$$H(A_i \oplus \Delta, \text{nonce}_{i,\text{even}}) \oplus \bigoplus_{j=0}^{2^i-1} S_{i,2j} \quad H(A_i, \text{nonce}_{i,\text{odd}}) \oplus \bigoplus_{j=0}^{2^i-1} S_{i,2j+1}$$

Note two facts: (1) each string $S_{i,j}$ is generated by invoking H on another uniform string and (2) both encryptions are generated by again invoking H. H is a circular correlation robust hash function, so $\mathcal{S}_{\text{obv}}$ securely simulates each pair of encryptions with uniform bits.

$\mathcal{S}_{\text{obv}}$ then copies the actions of OneHot.Ev in decrypting the GGM tree starting from the input shares. Recall that the purpose of this decryption is to compute the $2^n - 1$ leaf strings $L_{i \neq a}$. As an aside, we emphasize that the input a is not a uniform value, but rather must be simulated on a case-by-case basis. In particular, the Reveal leakage is simulatable (see below). Because each input share is uniform and each encryption is uniform, $\mathcal{S}_{\text{obv}}$ computes $2^n - 1$ uniform leaves.

Finally, in the real world, Gb appends to M m strings of the form:

$$\left(\bigoplus_i X_{i,j}\right) \oplus B_j$$

$\mathcal{S}_{\text{obv}}$ simulates each such string with a uniform string. This is suitable because (1) each $X_{i,j}$ is uniform, and (2) one string $X_{a,j}$ is missing from E's view. From here, $\mathcal{S}_{\text{obv}}$ copies the actions of Ev in computing E's share of the output matrix and outputs the resultant shares.

Thus, $\mathcal{S}_{\text{obv}}$ properly simulates one-hot gates.

Each recursive **Module call** is recursively simulated. This is appropriate by induction.

Reveal gates are handled as follows. First, we simulate shares for the sampled value $\alpha \leftarrow \mathcal{D}_{\text{mask}}$. These shares are trivially simulated by all zeros (see OneHot.Ev). Next, $\mathcal{S}_{\text{obv}}$ recursively simulates the Reveal gate's internal circuit C. By induction, this simulation is indistinguishable from the real garbling. Let b', c' be the simulated output shares. The real garbling appends a value Color(b) to the material (see OneHot.Gb). $\mathcal{S}_{\text{obv}}$ simulates this message as follows: first, it samples a value $x \leftarrow \mathcal{D}_{\text{out}}$. Recall that the user provided a proof that the output of C is indistinguishable from such a sampled value (Requirement 2). Hence x simulates the cleartext output of C. Next, $\mathcal{S}_{\text{obv}}$ appends $x \oplus \text{Color}(b')$ to its simulated material. This properly simulates the material because it "reveals" the value x, which is indistinguishable from the value revealed in the real garbling.

Color gates are straightforward: the simulator simply copies OneHot.Ev's actions.

We have now proved each gate type simulatable. However, this does not yet prove the entire module simulation indistinguishable. In the real world, some of the module's wires encode masks applied to values that are revealed via Reveal gates. We must ensure that these masks are never themselves revealed, or else the simulation would be distinguishable. This is ensured by three facts: (1) inside a module, no values are revealed except those revealed by new Reveal and Color gates, (2) each Reveal gate uses a fresh mask and its leakage is simulatable, and (3) the output of the module is a deterministic value, and hence is independent of any internal random masks (Requirement 1). Thus internal random masks cannot escape the module. Note also that the joint information given by multiple Reveal gates does not break indistinguishability (see Lemma 5.1).

Thus $\mathcal{S}_{\text{obv}}$ outputs a module garbling that is indistinguishable from real. Because modules can simulated, circuits can also trivially be simulated. OneHot is oblivious. □

We prove Theorem 5.7: If H is a circular correlation robust hash function, then OneHot is private.

Proof. By construction of a simulator $\mathcal{S}_{\text{prv}}$. At a high level, $\mathcal{S}_{\text{prv}}$ simply runs $\mathcal{S}_{\text{obv}}$, then builds a corresponding decoding string that ensures the simulated circuit garbling outputs y when evaluated.

First, $\mathcal{S}_{\text{prv}}$ invokes $(C, M', X') = \mathcal{S}_{\text{obv}}(1^\kappa, C)$. The remaining task is to generate a decoding string d' which, together with M' and X', is indistinguishable from real (M, X, d), even when given the output y.

To do so, $\mathcal{S}_{\text{prv}}$ invokes the procedure $Y' = \text{Ev}(C, M', X')$ and hence computes output shares that correspond to the obliviousness simulation. Recall that the real string d is constructed by hashing each corresponding zero/one output share (see OneHot.De). Thus, $\mathcal{S}_{\text{prv}}$ must simulate two strings for each ith output: one that properly maps Y_i' to y_i, and one that cannot be decrypted. $\mathcal{S}_{\text{prv}}$ computes $H(Y_i', \text{nonce})$ where nonce is the same nonce as described in OneHot.De. There are two available 'slots' in d' where this string can be placed; $\mathcal{S}_{\text{prv}}$ places it in slot y_i. $\mathcal{S}_{\text{prv}}$ fills the other slot with a uniform string.

Note first that the above is correct: OneHot.De$(d', Y') = y$. Moreover, the simulation is indistinguishable from the real world: each element in d is the output of a (circular correlation robust) hash function, so appears uniformly random; the simulated decoding string d' also appears uniformly random.

OneHot is private. □

We prove Theorem 5.9: If H is a circular correlation robust hash function, then OneHot is authentic.

Proof. We proceed backwards across C, at each gate demonstrating that $\mathcal{A}$ cannot obtain input shares except by correctly evaluating the previous parts of the circuit. The key idea is to show that forging an output of any subcircuit is as hard as forging an input to that subcircuit. Thus, by induction, forging a circuit output amounts to guessing a different circuit input, which succeeds with probability $2^{-\kappa}$ by trying to guess the value Δ. At a high level, authenticity is trivial for all except one-hot gates; one-hot gates are authentic due to the security properties of H.

First, inspect OneHot.De. Recall that for each bit of output y_i, the decoding string d holds two values:

$$H(\text{nonce}, Y_i) \qquad H(\text{nonce}, Y_i \oplus \Delta)$$

$\mathcal{A}$ succeeds if for any output bit y_i she causes De to output $y_i \oplus 1$. To construct an output that properly decodes, $\mathcal{A}$ must either (1) break the collision resistance of H (infeasible by assumption) or (2) construct a value $Y_i \oplus (y_i \oplus 1)\Delta$. If $\mathcal{A}$ attempts any other value, then OneHot.De will abort, so $\mathcal{A}$ fails.

Now, it suffices to show that it is infeasible to produce any such value $Y_i \oplus (y_i \oplus 1)\Delta$. We do so by induction on the structure of a module $\mathcal{M}$. Authenticity of circuits follows trivially from the authenticity of modules. The inductive hypothesis is as follows: Given garbled input X and material M for a submodule $\mathcal{M}'$ such that $Y = \text{Ev}(C, M, X)$, $\mathcal{A}$ cannot construct $Y' = Y \oplus r\Delta$ for some non-zero matrix r except with negligible probability. Put more simply, it is infeasible for $\mathcal{A}$ to generate an output Y' that is a valid share, but is different from Y anywhere.

XOR gates are trivially authentic. In Ev, XOR gates simply XOR the input shares. Thus, forging output is as hard as forging input.

Constant gates are trivially authentic: Ev simply outputs a share zero, so $\mathcal{A}$ is given no new information.

Recursive **Module calls** are authentic by induction.

Reveal gates are trivially authentic. First, each such gate samples a mask $\alpha \leftarrow \mathcal{D}_{\text{mask}}$. Forging a different mask would require $\mathcal{A}$ to guess Δ, which succeeds only with probability $2^{-\kappa}$ and hence is infeasible. Then, the gate forwards its input and α to a subcircuit which is authentic by induction.

Note, the Reveal gate also leaks a value to $\mathcal{A}$. Changing this value cannot help $\mathcal{A}$ because the leaked values are simply guides that indicate which garbled rows to decrypt.

Color gates are trivially authentic: Ev simply forwards parts of the Color gate input.

One-hot gates are less straightforward, but can be proved secure by the properties of H. Assume the one-hot gate has inputs a and b. We split the one-hot gate into two parts. The first part constructs the GGM tree where honest E decrypts 2^{n-1} out of 2^n leaves. The second part uses these leaves to construct matrix columns.

We start from the second part. For each column in the second part, the material includes a message of the following form:

$$\left(\bigoplus_i X_{i,j}\right) \oplus B_j$$

Moreover, $\mathcal{A}$ holds each string $X_{i \neq a,j}$ and $B_j \oplus b_j\Delta$. However, $\mathcal{A}$ does not hold $X_{a,j}$ and, moreover, this value is the result of calling H on a leaf string L_a. But H is a hash function, so constructing $X_{a,j}$ is as hard as constructing L_a. Thus, forging a valid output matrix $X' \neq X$ is as hard as forging L_a or forging valid $B' \neq B$.

Now, let us look at the first part and demonstrate that forging L_a is as hard as forging valid $A' \neq A$. We proceed upwards through the binary branching tree, demonstrating that forging level $i + 1$ is as hard as forging level i.

Each child node is constructed fby hashing its parent node, so forging a child is as hard as forging its parent. For each level, $\mathcal{A}$ additionally observes two strings:

$$H(A_i \oplus \Delta, \text{nonce}_{i,\text{even}}) \oplus \bigoplus_{j=0}^{2^i-1} S_{i,2j} \quad H(A_i, \text{nonce}_{i,\text{odd}}) \oplus \bigoplus_{j=0}^{2^i-1} S_{i,2j+1}$$

Informally, if $\mathcal{A}$ could decrypt the 'wrong' string, then she could forge a child on the path to a. However, both strings are encrypted using a hash of A_i and H is circular correlation robust. Therefore, forging the sum of odds/evens is as hard as forging $A' \neq A$. Thus, the first part is authentic.

In summary, forging an output matrix $X' \neq X$ is as hard as forging $A' \neq A$ or $B' \neq B$. Thus, one-hot gates are authentic.

Since all gate types are authentic, modules and circuits are also authentic. Forging a valid circuit output $Y' \neq Y$ is as hard as forging a valid circuit input $X' \neq X$, and forging a valid circuit $X' \neq X$ can only be achieved by guessing Δ, which only succeeds with negligible probability.

OneHot is authentic. □

A.1 Compatibility with Stacked Garbling

Recall that OneHot is a *garbling scheme* (Section 5.5, Construction 1) [BHR12]. In this section we prove OneHot's (limited) compatibility with stacked garbling [HK20a, HK21]. In particular, OneHot can serve as the *underlying* garbling scheme for stacked garbling, which handles each conditional branch. We start by giving the definition of *strong stackability* [HK21].

Definition A.1 (Strong Stackability). A garbling scheme is *strongly stackable* if:

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

(2) The scheme is *projective* [BHR12].

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

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

Informally, strong stackability achieves two goals. First, property (1) ensures that the garbling of a circuit "looks random", which is important when stacking branches [HK20a]. Second, properties (2–4) allow the stacked garbling scheme to manipulate the shares that emerge from evaluation of *our* garbling scheme.

To achieve strong stackability we modify OneHot in two ways.

The first change is simple and does not alter the flexibility of our scheme. Specifically, we alter our output decoding string d to meet item (3). Recall that we construct the decoding string d by setting the projective pair for each output bit y_i as follows (see Figure 4):

$$H(Y_i, \text{nonce}) \quad H(Y_i \oplus \Delta, \text{nonce})$$

Note that if we call our Color procedure (Section 3) on these two strings, the result may match, which fails property (3). Thus, we make the following simple adjustment to d:

$$H(\text{nonce}, Y_i) \mid \text{Color}(Y_i) \quad H(\text{nonce}, Y_i \oplus \Delta) \mid \text{Color}(Y_i \oplus \Delta) \quad (3)$$

By concatenating the color of the input shares, we ensure that the least significant bits of these two strings differ. Therefore, our Color procedure will now meet item (3).

The second change is more fundamental: Reveal gates can reveal values from *arbitrary* distributions. This breaks property (1), which insists that all values viewed by E are indistinguishable from uniform. Therefore, to achieve strong stackability we limit Reveal gates such that only uniform distributions are allowed.

Theorem A.2. *Let OneHot′ be the OneHot garbling scheme (Section 5.5, Construction 1) with the following two modifications:*

(1) *The output decoding string d is configured by setting the projective output pair for each output bit y_i according to Equation* (3).

(2) *The output distribution $\mathcal{D}_{\text{out}}$ of each Reveal gate is limited to a uniform distribution over binary strings.*

If H is a circular correlation robust hash function, OneHot′ is strongly stackable.

Proof. By inspection of the simulator $\mathcal{S}_{\text{obv}}$ (Theorem 5.5).

First note that strong stackability items (2–4) hold trivially. OneHot is projective. Color is formally defined in Section 3, and we define Key to be the procedure which drops the least significant bit (i.e. drops the color bit) and retains the remaining bits . The indistinguishability of pairs of keys in d follows from the fact that H is a circular correlation robust hash function (see Equation (3)).

Now, it remains to prove strong stackability item (1). Examine the obliviousness simulator $\mathcal{S}_{\text{obv}}$ (Theorem 5.5). Note that $\mathcal{S}_{\text{obv}}$ simulates the entire garbling (i.e., all material and wire shares) with uniform bits with one notable exception: the material used to reveal E's output of a Reveal gate is simulated by sampling from that gate's output distribution $\mathcal{D}_{\text{out}}$. However, since OneHot′ restricts $\mathcal{D}_{\text{out}}$, this simulation is also achieved by uniform bits. Since the simulator simulates all values with uniform bits, item (1) holds.

OneHot′ is strongly stackable. □

B APPLICATIONS – EXTENDED

In this appendix, we expand on details deferred from Section 5.

B.1 General Binary Outer Products – Extended

In Section 7.2 we explained our general outer product technique, but we did not formalize it. Figure 8 provides the formal module.

B.2 Binary Matrix Multiplication – Extended

It is well known that outer products can be used to efficiently compute matrix products. Specifically, the binary matrix product of input matrices a and b can be expressed by (1) for each i taking the outer product of column i of a with row i of b and (2) XORing the resulting matrices.

Notice that this technique does not use our low level primitives directly, and instead uses our outer product module as a black box. Hence, we need not formalize a module for matrix multiplication.

Because our technique reduces the cost of outer products by factor k (see Section 7.2), we similarly reduce the cost of binary matrix multiplication by factor k. That is, for input matrices with dimension $n \times m$ and $m \times l$, we require $O(nml/k)$ communication rather than the standard $O(nml)$. Formally, k is a logarithmic factor; in practice we instantiate k with small constants.

We implemented matrix multiplication; Figure 9 plots our improvement over the standard approach.

Input: Parties input shared bitstrings $[\![a]\!], [\![b]\!]$ where $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$.
Output: Parties output a shared matrix $[\![a \otimes b]\!]$.
Procedure:

- Parties release to E $a \oplus \alpha$ and $b \oplus \beta$ for uniform $\alpha \in \{0,1\}^n, \beta \in \{0,1\}^m$ via Color gates. Notably, the Color gates output $[\![a \oplus \alpha]\!]$, $[\![b \oplus \beta]\!]$, and $[\![\alpha]\!]$.
- Parties agree on a "chunk size" k which is at most logarithmic in the overall circuit input size. The parties split the input vectors into $\lceil n/k \rceil$ k-bit subvectors to avoid expensive exponential scaling.
- For each k-bit subvector $[\![a \oplus \alpha]\!]_{i..i+k}$, the parties compute:
  $$\mathcal{T}(\text{id})^{\mathsf{T}} \cdot [\![\mathcal{H}((a \oplus \alpha))_{i..i+k} \otimes b]\!] = [\![(a \oplus \alpha)_{i..i+k} \otimes b]\!]$$
  via a one-hot gate (by Lemma 4.1). Notice that the parties *do not* split b into chunks. The parties then vertically concatenate the $\lceil n/k \rceil$ resultant matrices into a single matrix $[\![(a \oplus \alpha) \otimes b]\!]$.
- Symmetrically, the parties compute $[\![(b \oplus \beta) \otimes \alpha]\!]$ by splitting $[\![b \oplus \beta]\!]$ into $\lceil n/k \rceil$ k-bit chunks.
- G locally computes $\alpha \otimes \beta$ and injects $[\![\alpha \otimes \beta]\!]$ as a constant.
- Parties compute and output:
  $$[\![(a \oplus \alpha) \otimes b]\!] \oplus [\![(b \oplus \beta) \otimes \alpha]\!]^{\mathsf{T}} \oplus [\![\alpha \otimes \beta]\!] = [\![a \otimes b]\!]$$
  (see Figure 5 for proof of above equality).

Figure 8: Efficient general outer product module. The module implements the function $a, b \mapsto a \otimes b$. Unlike Figure 5, this module handles outer products for input vectors of arbitrary length.

B.3 Integer Multiplication – Extended

Consider the multiplication of two n-bit numbers a and b. Standard GC techniques multiply such numbers using the schoolbook method [WMK16][6]. For sake of example, consider $n = 4$ and examine the computation done by the schoolbook method:

$$\begin{array}{rcccccc}
 & a_0 \cdot (& b_3 & b_2 & b_1 & b_0 &) \\
 & a_1 \cdot (& b_2 & b_1 & b_0 & 0 &) \\
 & a_2 \cdot (& b_1 & b_0 & 0 & 0 &) \\
+ & a_3 \cdot (& b_0 & 0 & 0 & 0 &) \\
\hline
 & & (ab)_3 & (ab)_2 & (ab)_1 & (ab)_0 &
\end{array}$$

Notice that each summand can be expressed by bits in the outer product of a and b. Hence, we improve multiplication by using our general outer product module (Section 7.2). Each summand must still be added inside GC; we do so by traditional GC means. This addition is now the bottleneck of multiplication performance. We leave potential improvements, perhaps by incorporating arithmetic GC techniques [BMR16], to future work.

Our integer multiplication technique does not use our lowest level primitives directly, so we need not formalize a module.

B.4 Binary Field Inverses – Extended

We now describe our field inverse module in detail. We first compute and release to E $a \cdot \alpha$. To do so, we use a Reveal gate to sample

[6] There exist asymptotically more efficient multiplication algorithms, but these are inefficient in practice.

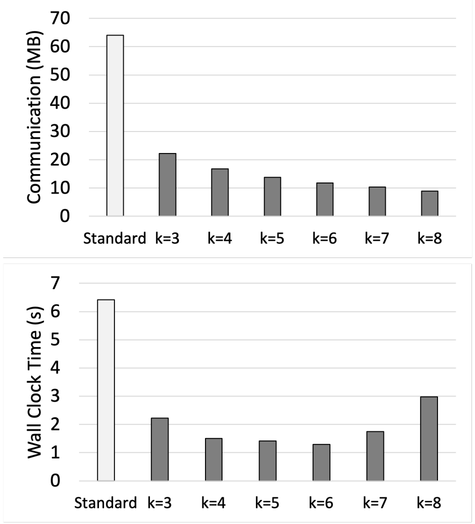

Figure 9: We used our implementation to compute the bitwise matrix product of two 128×128 square bit matrices. We plot total communication consumption (top) and wall clock runtime (bottom). We instantiated our approach with various "chunking factors" k (see Figure 8). At $k = 6$, we improve over standard by $6.2\times$ (communication) and $5\times$ (time).

uniform non-zero mask $\alpha \in \text{GF}(2^n)$. Recall that a Reveal gate may use an arbitrary circuit to apply the mask to its argument; we use a circuit that computes $a \cdot \alpha$ where $\cdot$ denotes field multiplication. This circuit uses the technique described in Section 7.5 to reduce field multiplication to an outer product. Because G knows the multiplicand α, the Reveal gate's internal circuit can compute the outer product more efficiently than as described in Section 7.1. Namely the parties compute the following, where $\gamma \in \{0,1\}^n$ is a uniform mask:

$$\mathcal{T}(\text{id})^\mathsf{T} \cdot [\![\mathcal{H}(a \oplus \gamma) \otimes \alpha]\!] \oplus [\![\gamma \otimes \alpha]\!] = [\![a \otimes \alpha]\!]$$

G injects $[\![\gamma \otimes \alpha]\!]$ as a constant. The parties apply a linear function to this outer product to obtain $[\![a \cdot \alpha]\!]$.

The above gate reveals $a \cdot \alpha$ to E, so we must prove this secure. This holds straightforwardly: because α is uniform and because $\text{GF}(2^n)$ is a field, this value is indistinguishable from a uniform non-zero field element.

Next, we use the revealed value $a \cdot \alpha$ to compute the inverse. Let $(\cdot)^{-1}$ be the function that takes the field inverse of its argument. The parties use a one-hot gate to compute the following:

$$\mathcal{T}((\cdot)^{-1})^\mathsf{T} \cdot [\![\mathcal{H}(a \cdot \alpha) \otimes \alpha]\!] = [\![(a \cdot \alpha)^{-1} \otimes \alpha]\!] \qquad \text{Lemma 3.3}$$

The parties use the reduction described in Section 7.5 to compute from the outer product the field product $[\![(a \cdot \alpha)^{-1} \cdot \alpha]\!] = [\![a^{-1}]\!]$.

Our module must account for the possibility that the input a is zero. The typical approach, which we also adopt, is to map input zero to output zero. To do so, we first compute an auxiliary bit z that indicates if $a = 0$. We use a regular Boolean circuit with ANDs and XORs to compute $[\![z]\!] \triangleq [\![a == 0]\!]$. At the top-level, the module computes the following expression:

$$[\![(a \oplus z)^{-1} \oplus z]\!]$$

If a is indeed zero, then this expression takes the inverse of one, which is itself one, and then XORs one, resulting in the desired output zero. Otherwise, this expression computes a^{-1}.

INPUT: Parties input shared bitstring $[\![a]\!]$ where $a \in \{0,1\}^n$.
OUTPUT: Let $f : \{0,1\}^n \to \{0,1\}^n$ be defined as follows:

$$f(x) \triangleq \begin{cases} 0 & \text{if } x = 0 \\ x^{-1} & \text{otherwise} \end{cases}$$

Parties output a shared bitstring $[\![f(a)]\!]$.
PROCEDURE:

- The parties first compute $[\![z]\!] \triangleq [\![a == 0]\!]$. This is achieved using a circuit with $n - 1$ AND gates.
- The parties next compute a Reveal gate to mask $a \oplus z$. The gate samples a uniform non-zero element α from $\text{GF}(2^n)$. The Reveal gate's internal circuit is itself a module that computes $x \mapsto x \cdot \alpha$; it is implemented as follows:
  - The internal module takes as arguments a and α. Via a Color gate, it reveals $a \oplus \gamma$ to E where γ is a uniform mask.
  - Note, G knows $\gamma \otimes \alpha$ and hence can inject it as a constant. The parties compute the following by a one-hot gate:
  $$\begin{aligned} &\mathcal{T}(\text{id})^\mathsf{T} \cdot [\![\mathcal{H}(a \oplus \gamma) \otimes \alpha]\!] \oplus [\![\gamma \otimes \alpha]\!] \\ &= [\![(a \oplus \gamma) \otimes \alpha]\!] \oplus [\![\gamma \otimes \alpha]\!] \\ &= [\![(a \otimes \alpha) \oplus (\gamma \otimes \alpha)]\!] \oplus [\![\gamma \otimes \alpha]\!] \\ &= [\![a \otimes \alpha]\!] \end{aligned}$$
  - The internal module then computes $[\![a \cdot \alpha]\!]$ via a linear function (see Section 7.5) and outputs the result.
- The above Reveal gate releases $(a \oplus z) \cdot \alpha$ to E. This is secure because both $a \oplus z$ and α are non-zero field elements and because α is uniform; hence the product is indistinguishable from a uniform non-zero field element.
- The parties compute the following by a one-hot gate:
$$\mathcal{T}((\cdot)^{-1})^\mathsf{T} \cdot [\![\mathcal{H}((a \oplus z) \cdot \alpha) \otimes \alpha]\!] = [\![((a \oplus z) \cdot \alpha)^{-1} \otimes \alpha]\!]$$
- Finally, the parties compute the following via a linear function (see Section 7.5) and output the result:
$$[\![((a \oplus z) \cdot \alpha)^{-1} \cdot \alpha) \oplus z]\!] = \begin{cases} [\![0]\!] & \text{if } z = 1 \\ [\![a^{-1}]\!] & \text{otherwise} \end{cases}$$

Figure 10: Our binary field inverse module.

B.4.1 S-Boxes – Extended. In Section 7.6, we discussed the 8-bit AES S-Box. 16-bit S-Boxes, based on an inversion in $\text{GF}(2^{16})$, have also been proposed for some applications [KKK+15]. The state-of-the-art Boolean circuit uses 226 ciphertexts (113 AND gates) [BMP13]. Our approach produces an S-Box that consumes only 122 ciphertexts, a ~ 45% improvement. Unfortunately, this application is less practical in terms of wall clock time since the parties must each compute a $2^{16} \times 16$ one-hot outer product matrix.

INPUT: Parties input shared bitstring $\llbracket a \rrbracket$ where $a \in \{0,1\}^n$. Parties agree on a public constant ℓ.
OUTPUT: Parties output a shared bitstring $\llbracket \ell^a \bmod 2^n \rrbracket$.
PROCEDURE:

- The parties use a Reveal gate to (1) sample uniform α, (2) compute $\llbracket a - \alpha \rrbracket$, and (3) reveal $a - \alpha$ to E. Because α is uniform, this revelation is secure.
- The parties view $\llbracket a - \alpha \rrbracket$ as the concatenation of $\lceil n/k \rceil$ k-bit "chunks". For each ith chunk $\llbracket c_i \rrbracket$, the parties compute:
$$\mathcal{T}(\ell^{(\cdot) \ll (i \cdot k)}) \cdot \llbracket \mathcal{H}(c_i) \otimes 1 \rrbracket = \ell^{c_i \ll (i \cdot k)}$$
- The parties compute and output:
$$\begin{aligned} &\left(\prod_i \ell^{c_i \ll (i \cdot k)} \right) \cdot \ell^{\alpha} \\ &= (\ell^{a-\alpha}) \cdot \ell^{\alpha} \\ &= \ell^a \end{aligned}$$
Note ℓ^{α} is a constant known to G. Each multiplication is computed via the technique described in Section 7.4.

Figure 12: Our integer exponent of a public constant module.

INPUT: Parties input shared bitstring $\llbracket a \rrbracket$ where $a \in \{0,1\}^n$. Parties agree on a public constant ℓ.
OUTPUT: Parties output a shared bitstring $\llbracket a \bmod \ell \rrbracket$.
PROCEDURE:

- Parties agree on a parameter m such that $m \cdot \ell > 2^n$.
- The parties use a Reveal gate to (1) sample uniform mask $\alpha \in \mathbb{Z}_{m \cdot \ell}$, (2) compute $\llbracket a + \alpha \bmod m \cdot \ell \rrbracket$, and (3) reveal $a + \alpha \bmod m \cdot \ell$ to E. Because α is uniform, this revelation is secure.
- The parties view $\llbracket a + \alpha \bmod m \cdot \ell \rrbracket$ as the concatenation of k-bit "chunks". For each ith chunk $\llbracket c_i \bmod (m \cdot \ell) \rrbracket$, the parties compute:
$$\begin{aligned} &\mathcal{T}(((\cdot) \ll (i \cdot k)) \bmod \ell) \cdot \llbracket \mathcal{H}(c_i) \otimes 1 \rrbracket \\ &= (c_i \ll (i \cdot k)) \bmod \ell \end{aligned}$$
Where $\ll$ denotes a left bit shift. That is, the parties compute $(\cdot) \bmod \ell$ on each k-bit chunk of the masked input.
- The parties compute and output:
$$\begin{aligned} &\left(\left(\sum_i (c_i \ll (i \cdot k)) \bmod \ell \right) + \alpha \right) \bmod \ell \\ &= ((a + \alpha) - \alpha) \bmod \ell \\ &= a \bmod \ell \end{aligned}$$
Each addition is computed by an AND-gate based circuit that efficiently computes $(x+y) \bmod \ell$ for x, y strictly less than ℓ.

Figure 11: Our public modular reduction module.

B.5 Modular Reduction – Extended

Figure 11 lists the module for our modular reduction technique. The module makes use of two key ideas:

First, consider x and y that are both statically known to be less than ℓ. In this case, the operation $(x + y) \bmod \ell$ is a special case and can be computed using linear communication: simply add the numbers, compare the sum to ℓ, and conditionally subtract ℓ.

Second, we use the two following equalities:

$$\begin{aligned} (x + y) \bmod \ell &= ((x \bmod \ell) + (y \bmod \ell)) \bmod \ell \\ x \bmod \ell &= (x \bmod (m \cdot \ell)) \bmod \ell \end{aligned}$$

Based on these ideas, we split the input a into chunks, reduce each chunk modulo ℓ, and then efficiently add the results.

B.6 Exponentiation – Extended

We now describe our module that computes $\llbracket \ell^a \rrbracket$ for publicly agreed constant ℓ. We take advantage of the following property of exponents:

$$x^y \cdot x^z = x^{y+z}$$

The module is formalized in Figure 12.

The key idea is that we can split a into "chunks" and then easily compute $\ell^{(\cdot)}$ for each chunk. We still need to then multiply each of these results inside GC, but the number of required multiplications is proportional only to the number of chunks. Notice also that we can use our improved multiplication technique (Section 7.4) to further improve the module.