Efficient Arithmetic in Garbled Circuits

David Heath(✉)

University of Illinois Urbana-Champaign, Champaign, USA
daheath@illinois.edu

Abstract. Garbled Circuit (GC) techniques usually work with Boolean circuits. Despite intense interest, efficient arithmetic generalizations of GC were only known from strong assumptions, such as LWE.

We construct symmetric-key-based arithmetic garbled circuits from circular correlation robust hashes, the assumption underlying the celebrated Free XOR garbling technique. Let λ denote a security parameter, and consider the integers $\mathbb{Z}_m$ for any $m \geq 2$. Let $\ell = \lceil \log_2 m \rceil$ be the bit length of $\mathbb{Z}_m$ values. We garble arithmetic circuits over $\mathbb{Z}_m$ where the garbling of each gate has size $O(\ell \cdot \lambda)$ bits. Contrast this with Boolean-circuit-based arithmetic, requiring $O(\ell^2 \cdot \lambda)$ bits via the schoolbook multiplication algorithm, or $O(\ell^{1.585} \cdot \lambda)$ bits via Karatsuba's algorithm.

Our arithmetic gates are compatible with Boolean operations and with Garbled RAM, allowing to garble complex programs of arithmetic values.

Keywords: Garbled Circuits · Arithmetic Circuits

1 Introduction

Yao's Garbled Circuit (GC) [25] is one of the main techniques for achieving secure multiparty computation (MPC). GC allows two parties, a garbler G and an evaluator E, to securely evaluate an arbitrary program over their joint private inputs. GC's crucial advantage is that it allows for protocols that run in only a constant number of rounds and that rely almost entirely on fast symmetric-key operations. Thus, GC is fast and flexible, making it a core tool in MPC.

While GC has steadily improved since Yao originally proposed the technique, GC still had a major weakness: arithmetic operations were expensive. This weakness is prominent because, by contrast, interactive secret-sharing-based MPC cost-efficiently generalizes from Boolean to arithmetic.

The Cost of GC. GC incurs three primary costs: (1) G's compute when garbling the program, (2) E's compute when evaluating the garbled program, and (3) the size of the garbled program. We refer to the bits of the garbled program as its *material*. The amount of material is the most interesting GC cost metric because it dictates communication cost, which is typically the performance bottleneck. We optimize GC material while keeping G's and E's compute reasonable.

Full version appears in IACR ePrint Archive: https://eprint.iacr.org/2024/139. This research was developed with funding from NSF grant CNS-2246353, and from USDA APHIS, under opportunity number USDA-APHIS-10025-VSSP0000-23-0003.

M. Joye and G. Leander (Eds.): EUROCRYPT 2024, LNCS 14655, pp. 3–31, 2024.
https://doi.org/10.1007/978-3-031-58740-5_1

Let λ denote the computational security parameter. Classic GC demonstrates how to garble any n-gate Boolean circuit with $O(n \cdot \lambda)$ bits of material. Prior to this work, efficient arithmetic generalizations of GC were not known, except those that only efficiently handle addition/subtraction [3] or that use heavy cryptographic assumptions such as learning with errors or decisional composite residuosity [1,2].

1.1 Contribution

We demonstrate efficient arithmetic garbled circuits from an assumption underlying most of the recent advances in symmetric-key-based GC.

Consider the integers $\mathbb{Z}_m$ for arbitrary modulus $m \geq 2$, and let $\ell = \lceil \log_2 m \rceil$. Let C be an n-gate arithmetic circuit over $\mathbb{Z}_m$ with addition, subtraction, and multiplication gates. Our garbling of C uses at most $O(n \cdot \ell \cdot \lambda)$ bits of material. Thus, each multiplication gate uses only $O(\ell \cdot \lambda)$ bits of material. Compare this with Boolean-circuit-based arithmetic, requiring $O(\ell^2 \cdot \lambda)$ bits via the schoolbook multiplication algorithm, or $O(\ell^{1.585} \cdot \lambda)$ bits via Karatsuba's algorithm [17]. Our scheme assumes only circular correlation robust hashes, the assumption underlying the popular Free XOR technique [8,19].

We consider two classes of moduli: arbitrary moduli m and *short* moduli 2^k for $k = O(\log n)$. While our handling of long moduli m is admittedly expensive, our handling of short moduli is surprisingly practical. Multiplication on short moduli 2^k costs only $(4k - 1) \cdot \lambda$ bits of material. Compare this to the $\approx (1.5 \cdot k^2) \cdot \lambda$ bits needed by state-of-the-art Boolean garbling [23] with schoolbook multiplication. Our short integer arithmetic becomes *even cheaper* when considering complex computations, such as vector inner products.

Our arithmetic values are compatible with Boolean operations. We can translate a value from arithmetic to Boolean (and vice versa) at cost $O(\ell \cdot \lambda)$ bits. This, for example, implies that comparisons are compatible with our approach. Comparisons are often a challenge for arithmetic systems.

We formalize our techniques in a novel model of computation that we call the *switch system* model. Switch systems describe computations as systems of equations. Switch systems generalize the recently proposed tri-state circuit model [15], a model that enables efficient garbling of RAM programs. Switch systems unify many GC capabilities, including Free XOR [19], the half-gates technique [26], our arithmetic techniques, the one-hot garbling technique [14], and Garbled RAM [20].

1.2 Background and Related Work

Basic Garbling. The basic idea underlying GC is to encode each input of a small function as *keys*, and then to use these keys to encrypt each row of the function's truth table. These keys are often called *labels*. We can design protocols that ensure that the evaluator E will only obtain labels that allow decryption of a *single* row, and – with care – this allows E to correctly evaluate any small function while remaining oblivious to the function input.

This idea sensibly extends from individual functions to Boolean circuits: Boolean gates are small functions, and we can use gate input labels to encrypt output labels. This basic approach leads to a garbling of size $O(n \cdot \lambda)$ [25]. While many works subsequently improved the handling of Boolean gates – e.g. [10,18,19,21–23,26] – asymptotic cost has not changed.

Challenge of Arithmetic Garbling. The natural generalization from the Boolean domain to arithmetic domains is impractical, because as we increase the bit-width of the domain, function tables grow in size exponentially. One might hope, then, that we can find ways to garble functions *without* encrypting function tables, and perhaps this would lead to arithmetic generalizations.

The celebrated Free XOR technique [19] achieves one such result. Free XOR allows us to garble XOR gates by simply XORing the gate's input labels. [3] showed that Free XOR indeed generalizes to arithmetic domains, so we can garble the addition operation of finite fields "for free".

While [3]'s arithmetic garbling can add arithmetic labels "for free", multiplication incurs *exponential* cost. Other works demonstrated better multiplication, but only by making strong assumptions, such as learning with errors, decisional composite residuosity, or bilinear maps [1,2,9]. Such approaches are of great interest, but they discard basic GC's practicality, and they require that we work with large numbers; as an example, [2] estimate their approach is similar in performance to the symmetric-key-based GC [23] once arithmetic values are almost *four thousand bits long.*

Compiling Arithmetic to Boolean. The practical approach to garbling arithmetic circuits did not use custom cryptography. It was better to simply compile each arithmetic gate to Boolean gates. To keep the resulting Boolean circuit's size in check, this approach leverages multiplication algorithms. In practice, GC uses either the classic schoolbook method (see e.g. [24]) or Karatsuba's algorithm [16,17].

Asymptotically efficient multiplication algorithms do exist, particularly the breakthrough $O(\ell \cdot \log \ell)$ multiplication of [13], but such algorithms involve infeasible constants. Thus, the reasonable approach to arithmetic GC used Karatusba's algorithm, incurring $O(\ell^{1.585})$ Boolean gates per multiplication. This superlinear circuit size made arithmetic operations expensive.

Figure 1 compares our approach with prior arithmetic techniques. In short, powerful and general techniques for arithmetic GC remained elusive.

One-Hot Garbling. The one-hot garbling technique [14] challenges the GC paradigm of encrypting truth table rows, and it supports efficient garbling of a new class of functions. Let $\mathbf{x}$ denote a length-n Boolean vector. Given a garbling of $\mathbf{x}$, the technique allows to efficiently compute a garbling of the *one-hot encoding* $\mathcal{H}(\mathbf{x})$, a length-2^n vector that holds zero at each index except index $\mathbf{x}$ (interpreting $\mathbf{x}$ as an integer), where it holds one.

Scheme	$+$	$\times$	$<$	Domain
This work (longs)	$O(\ell \cdot \lambda)$	$O(\ell \cdot \lambda)$	$O(\ell \cdot \lambda)$	$\mathbb{Z}_m$
This work (shorts)	$(2k-1) \cdot \lambda$	$(4k-1) \cdot \lambda$	$O(k \cdot \lambda)$	$\mathbb{Z}_{2^k}$
Schoolbook	$O(\ell \cdot \lambda)$	$O(\ell^2 \cdot \lambda)$	$O(\ell \cdot \lambda)$	$\mathbb{Z}_m$
Karatsuba [17]	$O(\ell \cdot \lambda)$	$O(\ell^{\log_2 3} \cdot \lambda)$	$O(\ell \cdot \lambda)$	$\mathbb{Z}_m$
CRT [1]	$O(\ell \cdot \lambda)$	$O(\ell \cdot \log \ell \cdot \lambda)$	✗	$\mathbb{Z}_N$
[3]	0	$O(2^\ell \cdot \lambda)$	$O(2^\ell \cdot \lambda)$	$\mathbb{Z}_p$
[3] w/ CRT	0	$O(\ell^2/\log \ell \cdot \lambda)$	$O(\ell^3/\log \ell \cdot \lambda)$	$\mathbb{Z}_N$

Fig. 1. Worst case GC material cost (in bits) of our approach as compared to other symmetric-key arithmetic GC that avoids practically infeasible algorithms. We highlight each column's most desirable asymptotic result. Low "$<$" cost demonstrates compatibility with Boolean operations. ℓ denotes the bit-length of values. m denotes an arbitrary modulus. p denotes a prime modulus. N is a product of pairwise coprime values, suitable for the Chinese Remainder Theorem (CRT, see Sect. 2.4).

These garbled one-hot encodings support two crucial operations. First, given a garbling of a bit y and a garbling of $\mathcal{H}(\mathbf{x})$, we can compute a garbling of the scaled vector $y \cdot \mathcal{H}(\mathbf{x})$ for only λ bits of material. This operation is limited in that it only works when the GC evaluator E knows $\mathbf{x}$ in cleartext, but in this case, it allows to compress the garbling of certain functions. Using basic GC, this operation would require $O(2^n \cdot \lambda)$ bits of material. Second, the one-hot garbling technique is compatible with Free XOR, meaning that for arbitrary affine function f, we can compute $f(\mathcal{H}(\mathbf{x}))$ for *no additional material*.

One-hot garbling is powerful because one-hot encodings are in a sense "fully homomorphic". Suppose we have a garbling of $\mathbf{x}$ where $\mathbf{x}$ is known to E, and suppose we wish to compute $f(\mathbf{x})$ for *arbitrary* f. [14] shows that we can (1) use $O(n \cdot \lambda)$ bits of material to compute $\mathcal{H}(\mathbf{x})$ and then (2) freely compute the linear operation $\langle \mathcal{T}(f) \cdot \mathcal{H}(x) \rangle$, where $\mathcal{T}(f)$ denotes the truth table of f and where $\langle \_ \cdot \_ \rangle$ denotes a vector inner product. This inner product "selects" the $\mathbf{x}$-th row of the truth table, computing a garbling of $f(\mathbf{x})$. Thus one-hot garbling can compute $f(\mathbf{x})$ for *any* f at material cost linear in $|\mathbf{x}|$.

Despite its power, one-hot garbling is limited: E must know $\mathbf{x}$ in cleartext, and (2) $\mathbf{x}$ must be relatively short, as the one-hot vector's length is exponential in $\mathbf{x}$'s length. When these constraints can be satisfied, the approach is useful.

One-hot garbling is a key ingredient in our approach. Indeed, we reconstruct the technique in our switch system formalism, and we demonstrate its compatibility with arithmetic values. We also optimize [14]'s technique, reducing by factor two the material cost to compute a one-hot encoding.

1.3 Summary of Our Approach

Our garbling of arithmetic circuits starts with a novel generalization of Free XOR [19]. Our new garbled labels encode integers in $\mathbb{Z}_{2^k}$ for any k. Unlike [3]'s generalization of Free XOR, our arithmetic labels are *not shorter* than a basic bit-by-bit garbling of an integer. However, our arithmetic labels have a crucial advantage over bit-by-bit garbling, because they allow us to add/subtract $\mathbb{Z}_{2^k}$ values "for free".

To multiply values, we demonstrate compatibility between our new labels and the one-hot garbling technique [14]. Let $x, y \in \mathbb{Z}_{2^k}$. We show how to multiply a garbled one-hot vector $\mathcal{H}(x)$ by an arithmetic label y yielding arithmetic label $x \cdot y \bmod 2^k$ for only $k \cdot \lambda$ bits of material.

We also give gadgets that *convert* between (1) garbled binary encodings of $\mathbb{Z}_{2^k}$ values, (2) garbled arithmetic encodings of $\mathbb{Z}_{2^k}$ values, and (3) garbled one-hot encodings of $\mathbb{Z}_{2^k}$ values. All such conversions cost at most $O(k \cdot \lambda)$ bits of material. By combining our multiplication procedure with conversions and one-time pad masks, we achieve arithmetic circuits over *short* integers, i.e. integers modulo $\mathbb{Z}_{2^k}$ where k is at most logarithmic in the circuit size.

To achieve arithmetic over *long* integers (i.e. integers modulo m for arbitrary m), we leverage the classic Chinese Remainder Theorem (CRT), which roughly states that arithmetic on long integers reduces to arithmetic on short integers. To complete the approach, we show that we can convert between long integers in binary representation and long integers in CRT representation. These conversions, again, heavily leverage one-hot garbling and our new arithmetic labels' free operations. With this done, we can handle arbitrary arithmetic circuits while using at most linear material per gate.

On Our Presentation. Much of our handling is intricate. For example, our conversion from an arithmetic label x to a one-hot encoding $\mathcal{H}(x)$ is tricky, as it requires that the GC evaluator E iteratively and simultaneously refine (1) a garbled binary encoding $\mathsf{bin}(x)$ and (2) a garbled one-hot encoding $\mathcal{H}(x)$. At each step, E uses one bit of $\mathsf{bin}(x)$ to solve for half of the remaining bits of $\mathcal{H}(x)$, then uses these new bits to solve for the next bit of $\mathsf{bin}(x)$, and so on. This iterative refinement boils down to solving a system of equations.

In light of this intricacy, our presentation is modular. First, we introduce a model of computation that we call the *switch system* model. This model captures our arithmetic operations, and it specifies computations as a system of constraints that E can solve. Proving that we can securely garble (oblivious) switch systems is relatively straightforward. With this done, we focus on switch systems and ignore garbling-specific concerns. We formalize our arithmetic techniques as switch systems, and this leads to a natural proof of security.

2 Preliminaries

2.1 Cryptographic Assumption

We use a circular correlation robust hash (CCRH) function H [8,26]. Roughly speaking, a CCRH produces random-looking output, even when hashing strings related by some correlation Δ, and even when using the output of the hash to encrypt strings that also involve the same correlation Δ. The CCRH definition enables the Free XOR technique [19], which leverages GC labels related by Δ. We use the CCRH definition given by [26]:

Definition 1 (Circular Correlation Robustness). *We define two oracles that each accept as input a label $K \in \{0,1\}^\lambda$, a nonce i, and a bit b:*

- $\mathsf{circ}_\Delta(K,i,b) \triangleq H(K \oplus \Delta, i) \oplus b \cdot \Delta$ *where* $\Delta \in \{0,1\}^{\lambda-1}1$.
- $\mathcal{R}(K,i,b)$ *is a random function with λ-bit output.*

A sequence of oracle queries (K,i,b) is legal *when the same value (K,i) is never queried with different values of b. H is **circular correlation robust** if no poly-time adversary $\mathcal{A}$ issuing legal queries can distinguish circ_Δ and $\mathcal{R}$. I.e.:*

$$\left|\Pr_{\Delta}\left[\mathcal{A}^{\mathsf{circ}_\Delta}(1^\lambda)=1\right] - \Pr_{\mathcal{R}}\left[\mathcal{A}^{\mathcal{R}}(1^\lambda)=1\right]\right| < \mathsf{negl}(\lambda)$$

In practice, H is often instantiated using fixed-key AES [11].

2.2 Garbling Schemes

A *garbling scheme* [7] is a tuple of procedures that specify how to garble a class of circuits.

Definition 2 (Garbling Scheme). *A **garbling scheme** for a class of circuits $\mathbb{C}$ is a tuple of procedures:*

$$\{\ \mathsf{Garble}, \mathsf{Encode}, \mathsf{Evaluate}, \mathsf{Decode}\ \}$$

where (1) Garble *maps a circuit $C \in \mathbb{C}$ to garbled circuit material $\hat{C}$, an input encoding string e, and an output decoding string d; (2)* Encode *maps an input encoding string e and a cleartext bitstring x to an encoded input; (3)* Evaluate *maps a circuit C, garbled circuit material $\hat{C}$, and an encoded input to an encoded output; and (4)* Decode *maps an output decoding string d and encoded output to a cleartext output string (or it outputs $\perp$ if the encoded output is invalid).*

A garbling scheme must be **correct** and may satisfy any combination of **obliviousness**, **privacy**, and **authenticity** [7]. The most interesting of these is obliviousness, which informally states that the garbled material together with encoded inputs reveals nothing to the evaluator:

Definition 3 (Oblivious Garbling Scheme). *A garbling scheme is* ***oblivious*** *if there exists a simulator* Sim *such that for any circuit* $C \in \mathbb{C}$ *and for all inputs* x *the following indistinguishability holds:*

$$(\tilde{C}, \text{Encode}(e, x)) \stackrel{c}{=} \mathsf{Sim}(1^\lambda, C) \qquad \textit{where } (\tilde{C}, e, \cdot) \leftarrow \text{Garble}(1^\lambda, C)$$

For most GC techniques (including ours), authenticity and privacy follow from obliviousness in a standard manner. Our full version expands.

2.3 Modular Arithmetic

We work with integers under various moduli, so we provide relevant notation.

- We use $[x]_m$ to denote the remainder of x divided by m.
- We use $x \equiv_m y$ to denote the modular congruence *relation*. Namely, x is congruent to – but not necessarily equal to – y.

When introducing variables we sometimes write $[x]_m$ to denote x is an integer modulo m. We also extend $[\cdot]_m$ notation to vectors. If $\mathbf{x}$ is a vector, then $[\mathbf{x}]_m$ denotes element-wise remainders:

$$[\mathbf{x}]_m = [\mathbf{x}[0], \ldots, \mathbf{x}[n-1]]_m \triangleq [\mathbf{x}[0]]_m, \ldots, [\mathbf{x}[n-1]]_m$$

We recall relevant properties of modular arithmetic:

$$[[x]_m + [y]_m]_m = [x+y]_m \qquad [[x]_m \cdot [y]_m]_m = [x \cdot y]_m$$
$$[[x]_{c \cdot m}]_m = [x]_m \qquad [x]_m \equiv_m x$$

2.4 Chinese Remainder Theorem

Our handling of long integers relies on the *Chinese Remainder Theorem* (CRT). Our full version reviews CRT, and it includes a formula that converts integers in CRT representation to integers in binary reprentation. We implement this conversion as part of our realization of long integer operations.

2.5 Barrett's Modular Reduction

One challenge in computing over a ring $\mathbb{Z}_m$ is simplifying values modulo m. The naïve approach – which implements modular reduction by repeated subtraction – uses $O(\ell^2)$ Boolean operations, which is too expensive.

Barrett [4] demonstrated another approach to modular reduction. Consider a value x that is less than m^2, sufficient to simplify products of values mod m. Barrett's approach computes $[x]_m$ using two multiplications, a division by a public power of four, and a *conditional subtraction*:

$$x \ominus y \triangleq \begin{cases} x - y & \text{if } x \geq y \\ x & \text{otherwise} \end{cases} \tag{1}$$

Barrett's approach reduces modulo m as follows:

$$[x]_m = \left(x - \left\lfloor \frac{x \cdot \left\lfloor \frac{4^\ell}{m} \right\rfloor}{4^\ell} \right\rfloor \cdot m \right) \ominus m \tag{2}$$

The crucial point of Equation (2) is that once we demonstrate a linear cost procedure for multiplying binary numbers, we obtain a linear cost procedure for multiplying numbers modulo m. This works because (1) division by public powers of four can be achieved by simply dropping least significant bits and (2) conditional subtraction can be implemented using well-known linear-sized Boolean circuits (see e.g. [24]). Therefore, it suffices to demonstrate a linear cost multiplication procedure for binary-encoded integers.

2.6 Miscellaneous Notation

- λ is a security parameter and can be understood as key length (e.g. 128 bits).
- 'msb' stands for 'most significant bit'; 'lsb' stands for 'least significant bit'.
- We use n to denote circuit size. We often consider values k that are at most logarithmic in n – i.e. $k = O(\log n)$ – such that 2^k is polynomial in n.
- We emphasize a value is a vector with bold: $\mathbf{x}$.
- $\mathbf{x}[0]$ is considered the msb of vector $\mathbf{x}$.
- We denote by $\mathbf{x} \sqcup \mathbf{y}$ the concatenation of $\mathbf{x}$ and $\mathbf{y}$.
- $x \triangleq y$ denotes that x is equal to y by definition.
- $x \overset{c}{=} y$ denotes that x is computationally indistinguishable from y.
- $x \leftarrow_{\$} D$ denotes that x is sampled from distribution D. If D is a set, we mean that x is drawn uniformly from D.

We discuss the *binary encoding* of integers:

Notation 1 (Binary Encoding)**.** *Let $x \in \mathbb{Z}_{2^k}$ be an integer and $\mathbf{x} \in \{0,1\}^k$ denote a vector. $\mathbf{x}$ is a **binary encoding** of x, written $\mathbf{x} = \mathsf{bin}(x)$, if:*

$$x = \mathsf{bin}^{-1}(\mathbf{x}) \triangleq \sum_{i \in [k]} 2^i \cdot \mathbf{x}[k-i-1]$$

3 Garbled Switch Systems

This section introduces our switch system model of computation and demonstrates how to garble any switch system. Sections 4 and 5 implement arithmetic operations in this model.

Switch systems are inspired by the tri-state circuit model [15], which was recently formalized as a basis for Garbled RAM [20]. We discuss connections between switch systems and tri-state circuits in the full version.

Our switch system model unifies many capabilities of GC, as it captures our arithmetic techniques, as well as many other GC techniques including Free XOR, One-Hot Garbling, and Garbled RAM.

3.1 Generalizing Free XOR

Our starting point is Free XOR [19]. In classic GC, the garbler G associates with each wire w two uniformly chosen labels, K_w^0 and K_w^1. At evaluation, E only holds the particular label associated with the logical value on each wire.

Free XOR changes the format of labels by sampling only *one* label K_w^0 per wire, and then defining K_w^1:

$$K_w^1 \triangleq K_w^0 \oplus \Delta$$

Here, G uniformly[1] draws $\Delta \in \{0,1\}^{\lambda-1}1$. Δ is *global* to the circuit, meaning that *each* pair of labels is correlated by the same value Δ.

The upshot is that G and E now implement XOR gates without G sending *any* material – the gate is "free". To garble an XOR gate $z \leftarrow x \oplus y$, G computes the labels for wire z based on the labels for x and y:

$$K_z^0 \triangleq K_x^0 \oplus K_y^0 \qquad K_z^1 \triangleq K_z^0 \oplus \Delta$$

When evaluating (and overloading the name of each input wire with its value), E holds labels K_x^x and K_y^y. E simply locally XORs the input labels, correctly computing an output label $K_x^x \oplus K_y^y = K_z^{x \oplus y}$.

Our Generalization of Free XOR. Consider a Free XOR label $K_x^x = K_x^0 \oplus x \cdot \Delta$, and view K_x^0, Δ as λ-bit *vectors*:

$$K_x^0 \oplus x \cdot \Delta = \begin{bmatrix} K_x^0[0] \\ \vdots \\ K_x^0[\lambda-1] \end{bmatrix} \oplus x \cdot \begin{bmatrix} \Delta[0] \\ \vdots \\ \Delta[\lambda-1] \end{bmatrix}$$

Viewed this way, there is a natural generalization from labels that encode *bits* to labels that encode *words*. Let $K_x^0, \Delta \in \mathbb{Z}_{2^k}^\lambda$ now denote vectors of $\mathbb{Z}_{2^k}$ elements[2], and let $x \in \mathbb{Z}_{2^k}$ now denote a word. We consider labels of the form $[K_x^0 + x \cdot \Delta]_{2^k}$:

$$[K_x^0 + x \cdot \Delta]_{2^k} = \left[\begin{bmatrix} K_x^0[0] \\ \vdots \\ K_x^0[\lambda-1] \end{bmatrix} + x \cdot \begin{bmatrix} \Delta[0] \\ \vdots \\ \Delta[\lambda-1] \end{bmatrix} \right]_{2^k}$$

Remark 1 (On the Size of Arithmetic Labels.). At first glance, it seems that our new labels have not given us anything new. To encode a k-bit word, we still require an encoding of length $k \cdot \lambda$, exactly as if we were to encode the word bit by-bit. However, our new labels enable new free operations, and these free operations ultimately enable efficient arithmetic circuits.

[1] Free XOR sets Δ's lsb to one, enabling the point-and-permute technique [6].

[2] In fact, we *could* generalize to *any* modulus m. We only consider moduli 2^k because they are sufficient and because this restriction allows security from Definition 1.

Free Operations on Arithmetic Labels. The first free operation is a direct generalization of Free XOR:

Lemma 1 (Free Affine Maps). *Let $f : \mathbb{Z}_m^s \to \mathbb{Z}_m^t$ denote an arbitrary affine map and let $\mathbf{x} \in \mathbb{Z}_m^s$ be a vector. Let $\otimes$ denote the* vector outer product *tensor:*

$$f([K^0_{\mathbf{x}} + \mathbf{x} \otimes \Delta]_{2^k}) = [f(K^0_{\mathbf{x}}) + f(\mathbf{x}) \otimes \Delta]_{2^k}$$

Proof. Immediate by the fact that f is affine. □

This implies that addition, subtraction, and multiplication by constants are free operations. For instance, to add two garbled words, just add the labels.

Our encoding also comes with a second free operation. Namely, we can in certain cases compute *modular reductions* for free. The following is immediate by properties of modular arithmetic:

Lemma 2 (Free Modular Reduction). *Let $x \in \mathbb{Z}_{2^{k+c}}$ be an integer.*

$$[[K^0_x + x \cdot \Delta]_{2^{k+c}}]_{2^k} = [K^0_x + x \cdot \Delta]_{2^k}$$

Said another way, if E holds a word label modulo 2^{k+c}, then E can freely simplify to a smaller word modulo 2^k by simply dropping msbs of each vector entry in the label. This allows us to extract labels encoding low bits of words, which is useful when converting from word labels to binary labels.

Remark 2 (Upcasting). Our modular reduction 'downcasts' large words to small words for free, but the other direction is not free. Our construction will require 'upcasting' binary labels to word labels, and this will require garbled material.

Finally, our word labels support a related operation that allows us to freely *discard* lsbs of labels, but only if they are known to be zero:

Lemma 3 (Free Division). *Let $x \in \mathbb{Z}_{2^{k+c}}$ be an integer such that $[x]_{2^c} = 0$. I.e., 2^c divides x. Integer division by 2^c is a free operation. Namely:*

$$\left\lfloor \frac{[K^0_x + x \cdot \Delta]_{2^{k+c}}}{2^c} \right\rfloor = \left[\left\lfloor \frac{K^0_x}{2^c} \right\rfloor + \frac{x}{2^c} \cdot \Delta \right]_{2^k}$$

Proof. Immediate by the fact that 2^c divides x. □

Crucially, if the above values K^0_x, Δ are uniform in $\mathbb{Z}^\lambda_{2^{k+c}}$, then $\lfloor K^0_x/2^c \rfloor$ (resp. $\lfloor \Delta/2^c \rfloor$) is uniform in $\mathbb{Z}^\lambda_{2^k}$. Floored division by 2^c partitions the values in $\mathbb{Z}_{2^{c+k}}$ into 2^k size-2^c congruence classes. This means that the above operation is safe in the sense that if the input label is uniform, then so is the output label. On the other hand, it is *not safe* to interpret the resulting quotient as an element modulo 2^{k+c}: the label $\lfloor K^0_x/2^c \rfloor$ is *not* uniform over $\mathbb{Z}^\lambda_{2^{k+c}}$.

The upshot is that if E is working with a word label encoding x, and if it is statically deducible that the lsb of x is zero, then E can locally discard the lsb of x by simply discarding the lsb of each of the λ vector entries in the label.

Multidirectional Gates. Free XOR labels – and by extension our arithmetic labels – allow E to compute addition gates by simply adding labels. As a thought experiment, suppose E instead is missing one gate input label, but has somehow obtained a gate output label. E can *solve* for the missing input label.

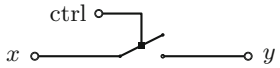
In other words, each XOR/addition expresses a linear relation on three labels, and given *any* two labels, E can solve for the third. Thus, we need not restrict ourselves to gates that evaluate only in one direction. This insight underlies our switch system model, which views garbled computation as taking place in a *constraint system*, where E uses some labels to solve for others, and where the order in which the system is solved might vary from one execution to another.

As we will see, our formalization of computations as constraint systems allow us to capture and improve the capabilities of one-hot garbling [14].

3.2 Switch Systems

We introduce our switch system model. In short, switch systems formalize the capabilities of our arithmetic labels (Sect. 3.1), making explicit various available operations, including our free operations.

Switch systems are circuit-like objects that establish constraints on arithmetic wires holding values under moduli 2^k for various k. As the name suggests, switch systems focus on components that we call *switches*:



A switch is a component relating three wires: a mod 2 *control* wire ctrl, and two mod 2^k *data* wires x and y. If the control wire holds logical zero, then the switch closes, connecting wires x and y such that they hold the same value; otherwise the switch remains open, and x and y are free to hold distinct values.

In the GC setting, we implement switches via a call to our CCRH function H. Namely, suppose wire x has zero label $[K_x^0]_{2^k}$ while Boolean wire ctrl has zero label $[K_{\mathsf{ctrl}}^0]_2$. G *defines* the zero label for wire y as follows:

$$K_y^0 \triangleq [K_x^0 + H(K_{\mathsf{ctrl}}^0, \nu)]_{2^k}$$

Here, ν is a nonce, and we ensure H outputs $\lambda \cdot k$ bits (by calling H k times).

If E knows the value of ctrl, and if $\mathsf{ctrl} = 0$, then E can use its ctrl label to compute $H(K_{\mathsf{ctrl}}^0, \nu)$, allowing E to compute the difference between the x and the y label such that E can indeed "connect" these wires. Note the inherent bidirectionality of the switch: E can use the difference between labels to translate an x label to y label, or vice versa.

Switches require that E know in cleartext the control wire's value. This is inherent from the fact that E's behavior is conditional, connecting two data wires iff $\mathsf{ctrl} = 0$. Of course, our goal is to handle arithmetic circuits that protect privacy, and so our handling must ultimately hide from E intermediate wire values. We achieve privacy-preserving computation via *oblivious* switch systems, which are explained later.

We now formalize the switch system model which captures the capabilities of our arithmetic representation. We show how to garble such systems shortly:

Definition 4 (Switch System). *A* ***switch system*** *is a system of constraints on wires (i.e., constrained variables) holding values over moduli* 2^k *for various* k*. The system is defined in terms of* gates*. Non-input wires in the system are initially* ***not set*** *(i.e., have no value), and as the system runs, wires become* ***set*** *according to the rules of each gate. The types of gates are as follows:*

- *A* ***switch*** *takes as input a binary* control wire $\mathsf{ctrl} \in \mathbb{Z}_2$ *and* data wire $x \in \mathbb{Z}_{2^k}$*. The gate outputs data wire* $y \in \mathbb{Z}_{2^k}$*, and it establishes the following implication constraint:*
$$\mathsf{ctrl} = 0 \implies x = y$$
We denote the output of a switch by writing $x \vdash \mathsf{ctrl}$.[3] *Switches are* bidirectional *in the sense that the system ensures that if* $\mathsf{ctrl} = 0$*, then* $x = y$*, regardless of which data wire is set first.*
- *A* ***join*** *takes input wires* $x, y \in \mathbb{Z}_{2^k}$ *and establishes an equality contraint:*
$$x = y$$
We denote a join by writing $x \bowtie y$*. Joins are* bidirectional *in the sense that the system ensures* $x = y$*, regardless of which wire is set first.*
- *An* ***affine gate*** *is parameterized by an affine map* $f : \mathbb{Z}_{2^k}^{\mathsf{in}} \to \mathbb{Z}_{2^k}^{\mathsf{out}}$*. It takes as input a vector of wires* $\mathbf{x} \in \mathbb{Z}_{2^k}^{\mathsf{in}}$*, and it outputs a vector of wires* $\mathbf{y} \in \mathbb{Z}_{2^k}^{\mathsf{out}}$*. The gate establishes the following constraint:*
$$f(\mathbf{x}) = \mathbf{y}$$
We denote affine gates by simply writing affine constraints of wires. Affine gates are multidirectional *in the sense that the system uses the values of set wires to solve for unset wires.*
- *A* ***modulus gate*** *takes as input a wire* $x \in \mathbb{Z}_{2^{k+c}}$ *for arbitrary* c, k*. It outputs a wire* $y \in \mathbb{Z}_{2^k}$*, and it establishes the following constraint:*
$$y = [x]_{2^k}$$
Modulus gates are one directional: the system uses x *to solve for* y*.*
- *A* ***division gate*** *takes as input a wire* $x \in \mathbb{Z}_{2^{k+c}}$ *where it is statically guaranteed that* 2^c *divides* x*. It outputs a wire* $y \in \mathbb{Z}_{2^k}$*, and it establishes the following constraint:*
$$y = [x/2^c]_{2^k}$$
Division gates are one directional: the system uses x *to solve for* y*.*

For convenience, we assume each gate has some unique identifier gid*. A switch system has* ***input wires*** *and* ***output wires****. For a switch system* S*, we denote by* $S(\mathbf{x})$ *the values on output wires after running with input wires* $\mathbf{x}$*.*

[3] $x \vdash \mathsf{ctrl}$ can be read 'x controlled by ctrl'. The symbol '$\vdash$' is meant to depict two vertical data wires connected at a point controlled by the horizontal control wire.

Remark 3 (Mismatched Moduli). Switch system joins/affine gates cannot combine values with different moduli. Indeed, corresponding operations on garbled words are only secure and correct when operating on matching moduli. Thus to, e.g., add a 1-bit value to a k-bit value, one must first "upcast" the 1-bit value to a k-bit value. Implementing such casts is a main challenge in our approach.

A crucial component of a switch system is its collection of control wires:

Definition 5 (Controls). *Let S be a switch system and let $\mathbf{x}$ be an assignment of input wires. The **controls of** S **on** $\mathbf{x}$, denoted $\mathsf{controls}(S, \mathbf{x}) \in \mathbb{Z}_2^*$, is the set of all switch control wire values (each labeled by its gate ID).*

As we discuss later, in a garbled switch system the controls of the switches are revealed to E; we introduce random masks as auxiliary input wires to ensure that all such revealed wires can be *simulated*.

Ruling Out Degenerate Systems. So far, it is not guaranteed that a switch system S with input $\mathbf{x}$ has only one possible configuration of wire values. For instance, using switches it is possible to introduce wires that, under particular inputs, are disconnected from the rest of the system. We are interested in well-formed systems where all wires are uniquely determined by the input wires:

Definition 6 (Legal Switch System). *A switch system S is **legal** if for any input $\mathbf{x}$, there exists only one assignment of circuit wires that satisfies the gate constraints. I.e., wire values are a function of the input wires.*

From here on, we only consider/construct legal switch systems.

Order of Gate Definition and the Need for Joins. It may seem strange that we specify gates as having inputs and outputs when gates are bidirectional.

We view such gates as producing their output wires because this is how the garbler G chooses wire labels. For example and as already discussed, for a switch gate, the label for wire y is computed from the labels for x and ctrl. Thus, we insist that switch system be written out as gates, each of which produces fresh output. This ensures G can compute all labels.

Joins provide a mechanism to connect two wires that are each the output of some gate. Joins may at first glance seem innocuous or even ad hoc. Not so. In our garbling, joins are the *only* significant source of garbled material. All other gates allow G and E to compute wire labels as a function of labels they already hold; joins require that G send to E the difference between two wire labels.

Crucially, switch system gates need not execute in the same order they are written down; E solves for unset wire labels as gate constraints become solvable.

Cost Metrics. The *size* of a switch system $|S|$ (the number of gates) is misleading as a cost metric, because switch systems allow wires over various moduli. Thus some wires carry more information than others, and, accordingly, some gates perform more work than others. To measure the complexity of a switch system, we measure the amount of information on wires. As we will see, the garbling of a

switch system grows only from join gates. Other gates are are 'free'. Accordingly, our most important metric is the total *width* of joined wires:

Definition 7 (Switch System Join Width). *Consider a switch system S, and let $x \in \mathbb{Z}_{2^k}$ denote a wire modulo 2^k in S. We say that wire x has **width** k. We denote the width of wire x by writing $\mathsf{width}(x)$. Consider a join $x \bowtie y$. The **width** of $x \bowtie y$ is the width of wire x (which is equal to the width of y):*

$$\mathsf{width}(x \bowtie y) \triangleq \mathsf{width}(x)$$

*The **join width** of S is defined by summing the width of each of S's join gates. We denote the join width of S by $\mathsf{join\text{-}width}(S)$. Width is measured in bits; we often say that S joins $\mathsf{join\text{-}width}(S)$ bits.*

Remark 4 (Reducing Garbled Material). As we will see, the amount of material needed to garble a switch system S is almost exactly $\mathsf{join\text{-}width}(S) \cdot \lambda$ bits. We can thus **reformulate our goal** of reducing material to reducing $\mathsf{join\text{-}width}(S)$.

Completeness. It may not be obvious that switch systems form a complete model of computation. To show that they are complete, and as a warm-up, we demonstrate switch systems are at least as powerful as Boolean circuits. (The following is similar to an argument about tri-state circuits [15]).

Theorem 1 (Emulating Boolean Circuits). *For any Boolean circuit C, there exists a switch system S such that:*

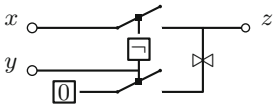
- $|S| = O(|C|)$ *and* $\mathsf{join\text{-}width}(S) = O(|C|)$.
- *For all inputs* $\mathbf{x}$, $S(\mathbf{x}) = C(\mathbf{x})$.

Proof. By constructing Boolean gates from switch system gates. More specifically, we emulate the complete Boolean basis $\{\oplus, \wedge, 1\}$.

Other than AND gates, emulation is straightforward: we can respectively emulate Boolean 0 and 1 by wire value $[0]_2$ and $[1]_2$, and each XOR is trivially emulated by an affine gate that adds values modulo 2 (i.e., computes XOR). Similarly, NOT gates can be emulated via an affine gate that adds $[1]_2$ to its input. AND gates are more complex, but can be emulated as follows:

$$\mathsf{AND}(x, y) \triangleq z \leftarrow x \vdash \neg y \quad ; \quad z \bowtie ([0]_2 \vdash y) \quad ; \quad \textbf{return } z$$

We sketch the emulated gate:



In our system, y and $\neg y$ are controls. If y holds 1, then only the top switch closes, connecting x to z; if y holds 0, then only the bottom switch closes, connecting 0 to z. Thus z is indeed the AND of x and y. We emphasize that we *must* include a join to properly set z. The system[4] joins only one bit. □

[4] In the terminology of [26], this system implements a "half gate", where E learns the cleartext bit y.

Oblivious Switch Systems. As already mentioned, our garbling of switch systems reveals to E all controls (Definition 5). Of course, our goal is to build garbling that preserves privacy, so we must ensure that the controls reveal no *useful* information. To achieve this, we consider switch systems with auxilliary random inputs; these random inputs will act as masks on cleartext values:

Definition 8 (Randomized Switch System). *A **randomized switch system** is a pair consisting of a switch system S and a distribution D. The execution of a randomized switch system on input $\mathbf{x}$ is defined by randomly sampling $\mathbf{r} \leftarrow_\$ D$, then running S on $\mathbf{x}$ and $\mathbf{r}$:*

$$(S, D)(\mathbf{x}) \triangleq S(\mathbf{x}; \mathbf{r}) \qquad \text{where } \mathbf{r} \leftarrow_\$ D$$

As we will see, we garble randomized switch systems by having G locally sample the distribution $\mathbf{r} \leftarrow_\$ D$; E does not know $\mathbf{r}$.

By including randomized inputs, we can consider switch systems that are *oblivious*, meaning that their control wires can be simulated:

Definition 9 (Oblivious Switch System). *Consider a* family *of legal randomized switch systems (S_i, D_i) for $i \in \mathbb{N}$. This family is **oblivious** if the distribution of controls (Definition 5) of (S_i, D_i) can be simulated. I.e., there exists a simulator $\mathsf{Sim}_{\mathsf{ctrl}}$ such that for all inputs $\mathbf{x}$:*

$$\mathsf{Sim}_{\mathsf{ctrl}}(1^\lambda) \stackrel{s}{=} \{\ \mathsf{controls}(S_\lambda, (\mathbf{x}; \mathbf{r}))\ \mid\ \mathbf{r} \leftarrow_\$ D_\lambda\ \}$$

Here, $\stackrel{s}{=}$ denotes that the distributions are statistically close (wrt λ).

Like deterministic switch systems, oblivious switch systems also form a complete model of computation. Namely, for any Boolean circuit, there is an oblivious switch system computing the same function.

Theorem 2 (Obliviously Emulating Boolean Circuits). *For any Boolean circuit C, there exists an **oblivious** switch system (S, D) such that:*

- $|S| = O(|C|)$ *and* $\mathsf{join\text{-}width}(S) = O(|C|)$.
- *For all inputs $\mathbf{x}$, $(S, D)(\mathbf{x}) = C(\mathbf{x})$.*

Proof. Theorem 1 shows that switch systems are complete, but the resulting construction is not oblivious. We achieve obliviousness via the 'half-gates' technique [26]. Namely, to AND bits x and y, we include in our randomized switch system's distribution D a Beaver multiplication triple [5]:

$$\{\ \alpha, \beta, \alpha \cdot \beta\ \mid\ \alpha, \beta \leftarrow_\$ \mathbb{Z}_2\ \}$$

Then, we use two non-oblivious AND gates (Theorem 1) to compute:

$$x \cdot (y \oplus \beta) \oplus \beta \cdot (x \oplus \alpha) \oplus \alpha \cdot \beta = x \cdot y$$

Because non-oblivious AND reveals its second argument to E, E learns $y \oplus \beta$ and $x \oplus \alpha$. α and β are uniform, so $y \oplus \beta$, $x \oplus \alpha$ can be simulated by uniform bits. Thus, the half-gates technique can be embedded in switch systems (the resulting GC material cost will match [26]'s 2λ bits per AND). □

We rely on Theorem 2 in parts of our construction of arithmetic circuits.

$\mathsf{Garble}(1^\lambda, (S_i, D_i))$:

$\Delta \leftarrow_\$ \mathbb{Z}^{\lambda-1}_{2^{\mathsf{max\text{-}width}}} \sqcup 1$

$\mathbf{r} \leftarrow_\$ D_\lambda$

for *each input* w *with width* k :

$K^0_w \leftarrow_\$ \mathbb{Z}^\lambda_{2^k}$

for *each random* w *set to* $[\mathbf{r}[i]]_{2^k}$:

$K^0_w \leftarrow [(0 - \mathbf{r}[i]) \cdot \Delta]_{2^k}$

for $(g, \mathsf{gid}) \in S_\lambda$; **match** g :

case $y \leftarrow x \vdash \mathsf{ctrl}$:

send $\mathrm{lsb}(K^0_{\mathsf{ctrl}})$

$K^0_y \leftarrow K^0_x + H(K^0_{\mathsf{ctrl}}, \mathsf{gid})$

case $x \bowtie y$: **send** $K^0_y - K^0_x$

case $\mathbf{y} \leftarrow f(\mathbf{x})$: $K^0_{\mathbf{y}} \leftarrow f(K^0_{\mathbf{x}})$

case $y \leftarrow [x]_{2^k}$: $K^0_y \leftarrow [K^0_x]_{2^k}$

case $y \leftarrow [x/2^c]_{2^k}$:

$K^0_y \leftarrow [K^0_x/2^c]_{2^k}$

$\mathsf{Sim}(1^\lambda, (S_i, D_i))$:

$\mathbf{ctrls} \leftarrow \mathsf{Sim}_{\mathrm{ctrl}}(1^\lambda)$

for *each input* w *with width* k :

$K_w \leftarrow_\$ \mathbb{Z}^\lambda_{2^k}$

for *each random* w : $K_w \leftarrow 0$

for $(g, \mathsf{gid}) \in S_\lambda$; **match** g :

case $y \leftarrow x \vdash \mathsf{ctrl}$:

send $\mathrm{lsb}(K_{\mathsf{ctrl}}) \oplus \mathbf{ctrls}[\mathsf{ctrl}]$

if $\mathbf{ctrls}[\mathsf{ctrl}] = 0$:

$K_y \leftarrow K_x + H(K_{\mathsf{ctrl}}, \mathsf{gid})$

else : $K_y \leftarrow_\$ \mathbb{Z}^\lambda_{2^k}$

case $x \bowtie y$: **send** $K_y - K_x$

case $\mathbf{y} \leftarrow f(\mathbf{x})$: $K_{\mathbf{y}} \leftarrow f(K_{\mathbf{x}})$

case $y \leftarrow [x]_{2^k}$: $K_y \leftarrow [K_x]_{2^k}$

case $y \leftarrow [x/2^c]_{2^k}$:

$K_y \leftarrow [K_x/2^c]_{2^k}$

Fig. 2. Our procedure for garbling oblivious switch system families (left) and our simulator used to prove security (right). $\mathsf{max\text{-}width}$ denotes the maximum width of any system wire. f ranges over affine functions. gid denotes a gate-specific nonce. lsb outputs the lsb of a Boolean GC label. H is a CCRH (Definition 1). **send** indicates to attach a string to the garbled material. The crucial security argument is that for each switch with a one control (highlighted), we can simulate the gate's output label by a uniform string.

3.3 Garbling Switch Systems

Our approach to garbling oblivious switch systems is relatively straightforward. We use H to implement switches, G sends lsbs of control wire labels to reveal their values, G sends differences between labels to implement joins, and all other gates are implemented via arithmetic label free operations (Sect. 3.1). G locally samples the oblivious distribution $\mathbf{r} \leftarrow_\$ D$; E does not learn $\mathbf{r}$. Crucially, the garbling of a switch system uses only $\approx \lambda \cdot \mathsf{join\text{-}width}(S)$ bits of material.

One key point is that G garbles gates in a fixed order, but E evaluates gates in whichever order it can. This order can vary with the system input, depending on which switches close and which do not. We formalize our handling:

Construction 1 (Garbled Switch Systems)**.** *We define our garbling scheme (Definition 2) for oblivious switch systems (Definition 9). In the following, each wire* w *has zero label* K^0_w*; we denote the runtime label held by* E *as* $K_w = K^0_w \oplus w \cdot \Delta$*. For simplicity, assume system input/output wires are mod* 2 *wires:*

- Garble *is defined in Fig. 2.*
- Encode *encodes each input wire value x as an input label $K_x = K_x^0 \oplus x \cdot \Delta$, where K_x^0 and Δ are chosen by* Garble*. Namely, our scheme's input encoding string e includes for each input wire the pair of possible labels.*
- Evaluate *uses available labels to solve for further labels:*
  - *Consider a switch $y \leftarrow x \vdash \mathsf{ctrl}$. Suppose K_{ctrl} and K_x (resp. K_y) are available.* Evaluate *uses the lsb included by* Garble *to decrypt* ctrl*. If* $\mathsf{ctrl} = 1$*, the gate is solved. If* $\mathsf{ctrl} = 0$*,* Evaluate *computes $H(K_{\mathsf{ctrl}}, \mathsf{gid})$ to find difference $K_y^0 - K_x^0$. It then adds (resp. subtracts) this difference to K_x (resp. K_y) to compute K_y (resp. K_x).*
  - *Consider a join $x \bowtie y$. Suppose K_x (resp. K_y) is available.* Evaluate *fetches the material $K_y^0 - K_x^0$ and adds (resp. subtracts) this difference to solve for K_y (resp. K_x).*
  - *Consider an affine gate $\mathbf{y} \leftarrow f(\mathbf{x})$, and suppose some set wires in $\mathbf{x}, \mathbf{y}$ fully determine some other wire.* Evaluate *uses affine operations on labels to solve for the determined wire's label.*
  - *Consider modulus gate $y \leftarrow [x]_{2^k}$.* Evaluate *drops msbs of entries of the vector K_x to compute $K_y \leftarrow [K_x]_{2^k}$.*
  - *Consider division gate $y \leftarrow [x/2^c]_{2^k}$ where x is statically guaranteed to be a multiple of 2^c.* Evaluate *drops lsbs of entries of the vector K_x to compute $K_y \leftarrow [K_x/2^c]_{2^k}$.*
- Decode *decodes each output wire label K_y as follows:*

$$\mathsf{Decode}(K_y) = \begin{cases} 0 & \text{if } H(K_y, \nu) = H(K_y^0, \nu) \\ 1 & \text{if } H(K_y, \nu) = H(K_y^0 \oplus \Delta, \nu) \\ \bot & \text{otherwise} \end{cases}$$

Here, K_y^0 and Δ are chosen by Garble*. Namely, our scheme's output decoding string d includes for each output wire the hash of the pair of possible output labels. Note, hashing the output labels ensures even a malicious evaluator cannot forge an output label that successfully decodes.*

Remark 5 (Revealing Control Bits). Construction 1 reveals control bits to E by including in the GC material lsbs of control wires. These are individual bits, not length-λ strings. Sending a bit for *every* switch is overkill, as the value of one control bit is often deducible from other control bits. Rather than meticulously accounting for this, we simply point out that in our constructions the number of control bits that need to be revealed is small, and is not asymptotically relevant. From here on and when counting material cost, we only count joined bits, which are *significantly* more expensive than revealed control bits.

Construction 1 is correct, and it is also oblivious (Definition 3) so long as the switch system is itself oblivious (Definition 9). Accordingly, the scheme can be used to build GC protocols.

Theorem 3 (Obliviousness). *When (S_i, D_i) is an oblivious switch system family, Construction 1 is an oblivious garbling scheme. Namely, let H be a circular correlation robust hash function (Definition 1). There exists a simulator* Sim *such that for all inputs* $\mathbf{x}$ *the following indistinguishability holds:*

$$(\tilde{S}, \text{Encode}(e, \mathbf{x})) \stackrel{c}{=} \mathsf{Sim}(1^\lambda, (S_\lambda, D_\lambda)) \quad \textit{where } (\tilde{S}, e, \cdot) \leftarrow \text{Garble}(1^\lambda, (S_\lambda, D_\lambda))$$

The full version provides a detailed proof of Theorem 3. For now, we provide a detailed obliviousness simulator and sketch an argument of security.

Proof Sketch By construction of a simulator Sim (Fig. 2). In short, Sim produces a convincing view by using (1) the properties of H and (2) the switch system's control wire simulator $\mathsf{Sim}_{\mathsf{ctrl}}$.

The real garbling of (S_λ, D_λ) reveals to E the control bits $\mathsf{controls}(S_\lambda, (\mathbf{x}; \mathbf{r}))$, and this string depends on the input $\mathbf{x}$. Sim does not know $\mathbf{x}$, so it cannot reveal the same values. Instead, it uses the obliviousness of (S_i, D_i) to call $\mathsf{Sim}_{\mathsf{ctrl}}$, allowing it to reveal values that are *statistically close* to the real world controls.

The remaining challenge is to simulate output labels from switches. For each switch, there are two cases. If the control holds zero (the switch is closed), Sim matches the real-world garbling. If the control holds one, Sim simulates the label with a uniformly random string; this is a good simulation because of the properties of the CCRH and because we know E will not learn the control wire zero label. There is some nuance in showing CCRH is sufficient to garble an inactive switch, since CCRH is defined for Boolean strings, but our labels are $\mathbb{Z}_{2^k}$ vectors. Resolving this mismatch is not hard; see the full proof for details. □

4 Generalized One Hot Garbling

The core of our approach uses switch systems to connect our arithmetic labels (Sect. 3.1) with the one-hot garbling technique [14]. Our handling of arithmetic circuits ultimately reduces to switch systems developed in this section.

Our new one-hot formalism is more efficient than the presentation of [14]. [14] uses $2(n-1) \cdot \lambda$ bits of material to garble a one-hot encoding (defined shortly) of a length-n string; we improve by factor two, achieving the same result at cost $(n-1) \cdot \lambda$ bits. Unwinding our handling of switch system gates, the following approach is similar to the GGM tree improvement of [12].

We start by defining one-hot encodings. The one-hot encoding of an integer x is a zero/one vector that – as the name suggests – is *one-hot*: it has exactly one non-zero entry, and the location of this entry encodes x:

Notation 2 (One-Hot Encoding)**.** *Let $x \in \mathbb{Z}_{2^k}$ be an integer. The* ***one-hot encoding*** *of x is a length 2^k vector $\mathcal{H}(x)$ s.t. each $\mathcal{H}(x)[i]$ is a one iff $x = i$:*

$$\mathcal{H}(x) = \bigsqcup_{i \in [2^k]} (x \stackrel{?}{=} i)$$

Remark 6 (Arithmetic and Binary One-Hot Encodings). It will be convenient to consider one-hot encodings both where zero/one entries are in $\mathbb{Z}_2$ and where zero/one entries are in $\mathbb{Z}_{2^k}$. We refer to the former as a *binary* one-hot encoding and to the latter as an *arithmetic* one-hot encoding.

The crucial property of one-hot encodings is that they are 'fully homomorphic' in the sense that we can evaluate an arbitrary function via an affine map:

Lemma 4 (Evaluation via Truth Table [14]). *Let $x \in \mathbb{Z}_{2^k}$ and $f : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ denote a function.*

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

Here, entries of $\mathcal{H}(x)$ are $\mathbb{Z}_{2^k}$ elements, $\mathcal{T}(f)$ denotes the truth table *of f expressed as a vector, and $\langle \llcorner \cdot \lrcorner \rangle$ denotes the vector inner product operation.*

Thus, if we construct a one-hot encoding of a value x, then we can compute $f(x)$ for free. This is crucial throughout our approach.

Remark 7 (Obliviousness). In the remainder of this section, assume that all one-hot positions are known in the clear to the evaluator E. Our handling of arithmetic circuits later uses one-time pads to mask true values from E, achieving oblivious switch systems (Definition 9) and thus secure arithmetic GC.

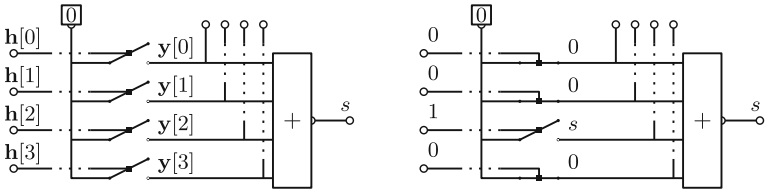
4.1 Our Approach to One-Hot Garbling

Our first goal is to construct a switch system that on input a binary encoding $\mathbf{x} = \mathsf{bin}(x)$ outputs a binary one-hot encoding $\mathcal{H}(x)$. We approach this problem recursively, so assume that we have a one-hot encoding of the first i bits of $\mathbf{x}$; we wish to construct a one-hot encoding of the first $i+1$ bits.

Our key tool for taking this step is a switch system that scales a one-hot vector $\mathbf{h}$ by some scalar $s \in \mathbb{Z}_{2^k}$ for arbitrary k:

$$\mathsf{scale}_k(\mathbf{h} = \mathcal{H}(x)) \triangleq \mathbf{y} \leftarrow \left(\bigsqcup_i [0]_{2^k} \vdash \mathbf{h}[i] \right) \;;\; s \leftarrow \sum_i \mathbf{y}[i] \;;\; \textbf{return } (s, \mathbf{y})$$

The system treats scalar s as an output, not as an input; this will be formally convenient later. As wires are bidirectional, s can be 'converted to an input' by joining it with some other wire. For reference, we draw an example of the above system both with symbolic inputs (left) and on a concrete input (right):



We construct each word of the output vector $\mathbf{y}[i]$ by switching data value $0 \in \mathbb{Z}_{2^k}$ with control $\mathbf{h}[i]$. Because $\mathbf{h}$ is one-hot, all except one switch closes. We add together the entries of $\mathbf{y}$, and we name the sum s.

Since all except one wire in $\mathbf{y}$ holds a zero, there is a unique wire assignment that satisfies the system: the single non-zero entry of $\mathbf{y}$ must be equal to s. In the context of garbling, this means that E can use available zero labels and a label that encodes s to solve for the single non-zero label in the encoding of $\mathbf{y}$. Thus, this system indeed scales the one-hot vector $\mathbf{h}$ by the scalar s. The scale system has no join gates, and hence it is "free".

We can use scale to construct a one-hot encoding of the binary vector $\mathbf{x}$ recursively. In the base case, the one-hot encoding of a single bit $\mathbf{x}[0]$ is simply the pair $(\neg\mathbf{x}[0], \mathbf{x}[0])$. In the recursive case, we take the one-hot encoding of the first i bits of $\mathbf{x}$ and scale it by the next bit of $\mathbf{x}$. From here, we extend the one-hot encoding with XORs (addition mod 2). The construction is as follows:

$$
\begin{array}{l}
1\ \ \mathsf{bin\text{-}to\text{-}hot}(\mathbf{x} = \mathsf{bin}(x)) \triangleq \\
2\ \ \ \textbf{if}\ (|\mathbf{x}| = 1): \quad \textbf{return}\ (\neg\mathbf{x}[0], \mathbf{x}[0]) \\
3\ \ \ \textbf{else}: \\
4\ \ \ \ \mathbf{h} \leftarrow \mathsf{bin\text{-}to\text{-}hot}(\mathbf{x}[1..]) \ \ ; \ \ (s, \mathbf{h}') \leftarrow \mathsf{scale}_1(\mathbf{h}) \ \ ; \ \ s \bowtie \mathbf{x}[0] \\
5\ \ \ \ \textbf{return}\ ([\mathbf{h}' + \mathbf{h}]_2) \sqcup \mathbf{h}'
\end{array}
$$

Given a length-n input vector, each recursive call joins one bit, so $\mathsf{bin\text{-}to\text{-}hot}$ joins $n-1$ total bits. Again, this is a factor two improvement over [14].

4.2 Half Multiplication

As a stepping stone to full multiplication of short arithmetic values, we build a 'half multiplier', similar in flavor to [26]'s half AND gate (see Theorem 1). The system takes as input one-hot-encoded x and word-encoded $y \in \mathbb{Z}_{2^k}$:

$$\mathsf{half\text{-}mul}(\mathbf{h} = \mathcal{H}(x), y) \triangleq (s, \mathbf{h}') \leftarrow \mathsf{scale}_k(\mathbf{h})\ ;\ s \bowtie y\ ;\ \textbf{return}\ \sum_i i \cdot \mathbf{h}'[i]$$

$\mathsf{half\text{-}mul}$ observes that we can scale x's one-hot vector by y, then use affine operations to scale each entry of $\mathbf{h}'[i]$ by its index i (each i is a constant). After this, the one-hot location stores $[x \cdot y]_{2^k}$, and all other locations store zero. Summing these products computes $[x \cdot y]_{2^k}$. $\mathsf{half\text{-}mul}$ joins only k bits.

4.3 Conversions

To support general arithmetic operations, we need a variety of *conversion* operators that move between various data representations.

Conversions from One-Hot Encodings. It is useful to convert an arithmetic one-hot encoding $\mathcal{H}(x)$ to binary encoding $\mathsf{bin}(x)$ and/or word encoding x. These conversions are free, due to the 'fully homomorphic' nature of one-hot encodings.

In more detail, let $\mathsf{id} : \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ denote the identity function.

$$\mathsf{hot\text{-}to\text{-}word}(\mathbf{h} = \mathcal{H}(x)) \triangleq \langle \mathcal{T}(\mathsf{id}) \cdot \mathbf{h} \rangle$$

By Lemma 4 this correctly computes the word encoding of x.

Similarly, we can extract a binary encoding of an arithmetic one-hot encoded value $\mathcal{H}(x)$. We simplify the one-hot indices modulo 2, then take an appropriate linear combination of the wires:

$$\mathsf{hot\text{-}to\text{-}bin}(\mathbf{h} = \mathcal{H}(x)) \triangleq \langle \mathcal{T}(\mathsf{bin}) \cdot [\mathbf{h}]_2 \rangle$$

Converting Words to One-Hots. We can now convert binary strings to one-hot vectors, and we can easily convert one-hot vectors to words/binary. The final conversion – words to one-hot vectors – is significantly more complex.

Our observation is that to compute a one-hot encoding of x, we first need a *binary* encoding $\mathsf{bin}(x)$. If we can achieve this, then we can apply $\mathsf{bin\text{-}to\text{-}hot}$, scale the result by $[1]_{2^k}$, and the problem is solved. However, efficiently converting a word x to a binary encoding $\mathsf{bin}(x)$ is non-trivial.

Our mod gate allows us to freely extract from x the lsb $[x]_2$. However, higher bits are harder to extract. Our division gate *would* allow us to make progress, *if* we could ensure that the lsb of x was zero. Of course, x might have lsb one, so this does not yet work. To extract the second lsb of x, we need to first *subtract off* the lsb. We have this lsb $[x]_2$, but we cannot yet subtract it off, because its modulus 2 does not match x's modulus 2^k, and our linear operations require wires with matching moduli (Remark 3). Thus, to compute $[x - [x]_2]_{2^k}$, we first need to "upcast" the bit $[x]_2$ to a word $[[x]_2]_{2^k}$.

As a strawman solution to this upcast problem, consider the following system:

$$([0]_{2^k} \vdash [x]_2) \bowtie ([1]_{2^k} \vdash \neg[x]_2)$$

This strawman is correct: if $[x]_2 = 0$, then the joined wire hold $[0]_{2^k}$; otherwise, the joined wires hold $[1]_{2^k}$. *The problem* is that this system joins k bits, which is simply too expensive. To extract *all* bits of x, we would need to upcast *each* of its bits in turn, and each upcast would join $O(k)$ bits. In total we would join $O(k^2)$ bits, which is useless for linear cost arithmetic.

Still, the above template ultimately leads to an efficient solution: (1) use mod to obtain the lsb of x, (2) upcast the lsb to a word, (3) subtract the upcasted lsb from x, (4) now that the lsb of x is guaranteed to be zero, use division to remove the lsb, and (5) repeat. To instantiate this template efficiently, our system upcasts all of the lsbs of x *in batch*. The resulting solution is a single switch system that converts each bit of x to binary, and it simultaneously converts x to arithmetic one-hot representation. The full system joins only $2k - 1$ bits.

```
1  word-to-hot_k(x) ≜
2    h_bin ← bin-to-hot(bin(x))  ;  h_arith ← ⨆_i ([0]_{2^k} ⊢ h_bin[i])
3    (h^ℓ, h^r) ← halves(h_arith)  ;  bin(x) ← solve-bin_{k−1}(h^ℓ + h^r)
4    return h_arith
5      where solve-bin_i(h = H([x]_{2^i})) ≜
6        if (i = 0) :  h[0] ⋈ [1]_{2^k}  ;  return [x]_2
7        else :
8          [[x]_{2^i}]_{2^k} ← hot-to-word(h)  ;  msb ← [(x − [x]_{2^i})/2^i]_2
9          (h^ℓ, h^r) ← halves(h)  ;  lsbs ← solve-bin_{i−1}(h^ℓ + h^r)
10         return msb ⊔ lsbs
```

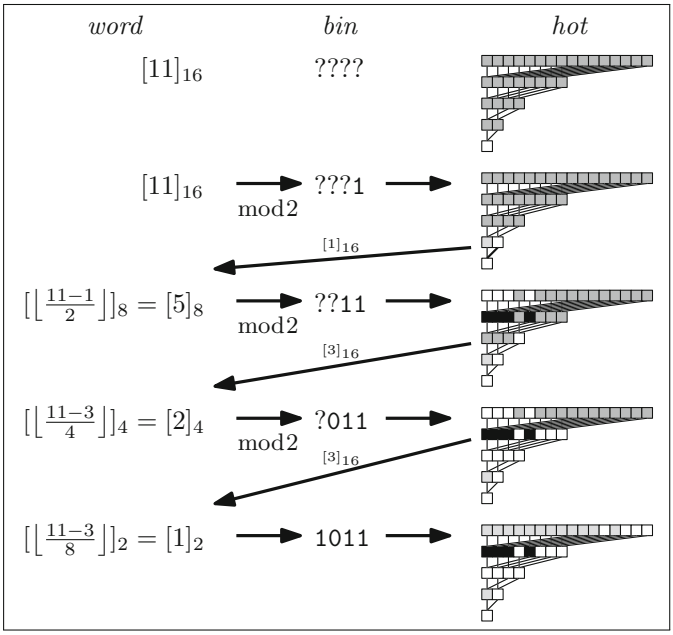


Fig. 3. (Top) our word-to-hot switch system converts an arithmetic word $[x]_{2^k}$ to an arithmetic one-hot vector $[\mathcal{H}(x)]_{2^k}$. halves splits a length-2^i vector at its middle, giving two length-2^{i-1} outputs. (Bottom) an example word-to-hot execution. The system maintains three encodings of its input x: a word encoding, a binary encoding, and an arithmetic one-hot encoding (as well as one-hot encodings of x modulo powers of two). Black squares represent logical zero; white squares represent one; gray squares represent an unknown value. At each step, the system uses mod to extract the lsb of x, yielding the next bit in the binary encoding. It then can refine its one-hot encodings. These one-hot encodings allow to linearly compute a word encoding of lsbs, which are subtracted from x, allowing us to repeat until we fully solve the one-hot encoding.

word-to-hot (see Fig. 3) is our most intricate construction. It leverages gate bidirectionality to solve for x's binary encoding and one-hot encoding *together*. Figure 3 sketches an example, and it show how E solves the constraints.

The system itself is circularly defined: the binary one-hot vector $\mathbf{h}_{\mathsf{bin}}$ is defined based on $\mathsf{bin}(x)$, which is not defined until the later call to solve-bin. This definition is nevertheless sensible: the system iteratively *refines* its solution for both $\mathsf{bin}(x)$ and $\mathbf{h}_{\mathsf{bin}}$. Note, the system can immediately solve for the lsb of x (base case of solve-bin) via a mod gate.

In a sense, we construct $\mathbf{h}_{\mathsf{bin}} = [\mathcal{H}(x)]_2$ by *imagining* that we already have $\mathsf{bin}(x)$. With $\mathbf{h}_{\mathsf{bin}}$ constructed, we use switches to cast the zero slots of $\mathbf{h}_{\mathsf{bin}}$ to words mod 2^k, yielding an arithmetic one-hot vector $\mathbf{h}_{\mathsf{arith}}$ whose one slot is initially unconstrained. We again *imagine* that this one slot is filled with $[1]_{2^k}$, in which case $\mathbf{h}_{\mathsf{arith}}$ would be a valid arithmetic one-hot encoding of x. Then, we repeatedly take left and right halves of $\mathbf{h}_{\mathsf{arith}}$ and sum them to a vector of half the length. This iteratively yields shorter and shorter arithmetic one-hot vectors:

$$[\mathcal{H}([x]_{2^k})]_{2^k} \quad [\mathcal{H}([x]_{2^{k-1}})]_{2^k} \quad \ldots \quad [\mathcal{H}([x]_2)]_{2^k} \quad [1]_{2^k}$$

The final singleton one-hot vector is degenerately equal to $[1]_{2^k}$, but only because we *imagined* that the one slot of $\mathbf{h}_{\mathsf{arith}}$ was filled, so in fact this final value is so far *unconstrained*. We *insist* that this final value is indeed $[1]_{2^k}$ by applying a join gate (base case of solve-bin).

If we can indeed solve for the one-hot encoding of x, then we can solve for intermediate arithmetic one-hot encodings by propagating $[1]_{2^k}$ *backwards* through the system (see Fig. 3). From here, we use the intermediate encodings to linearly compute word encodings of each bit of x.

Evaluation of the system proceeds as follows. Because the lsb of x is initially available, E can solve for *half* of the entries of $\mathbf{h}_{\mathsf{bin}}$. Specifically, E solves for each one-hot index whose lsb does *not* match that of x; see Fig. 3. This, in turn, lets E solve for a word encoding of x's lsb, which can be subtracted from x. Now x must have lsb zero, so we can iterate, extracting a binary encoding of x's second lsb, allowing E to refine the one-hot encoding of x, allowing E to obtain a word encoding of x's second lsb, and so on.

While intricate, word-to-hot is lean: it joins only $2k - 1$ bits. This low cost makes it affordable to convert arithmetic labels to arithmetic one-hot encodings.

5 Garbled Arithmetic from Switch Systems

We now have our core tools that combine one-hot garbling with arithmetic labels. From here, we assemble arithmetic garbling. We start by building switch systems for short integers; this handling is straightforward and practical. We use short integers to achieve long integers by applying the Chinese Remainder Theorem.

5.1 Short Integers

Based on our new one-hot operations (Sect. 4), we can construct oblivious switch systems (Definition 9) that operate on *short* integers (henceforth, "shorts"). Here, we consider integers modulo 2^k where $k = O(\log n)$ is at most logarithmic in the circuit size n. The fact that k is small allows us to write out one-hot encodings of integers in polynomial time. Later, we will use shorts as a building block to garble *long* integers, which have no restriction on the modulus.

Notation 3 (Short). *The **short encoding** of integer $x \in \mathbb{Z}_{2^k}$ – written* $\mathsf{short}(x)$ *– consists of (1) a uniform mask $\alpha \in \mathbb{Z}_{2^k}$ and (2) a length 2^k arithmetic one-hot encoding $\mathcal{H}(x+\alpha)$.*

Recall that in a one-hot encoding, the evaluator E knows the one-hot value in cleartext; the one-time pad mask α hides x. In the context of a randomized switch system, the mask α is part of the system's randomized inputs, and in GC the garbler G knows α in cleartext.

Conversions on Shorts. Before explaining how to add/subtract/multiply shorts, we need conversion operations on shorts. These operations are a straightforward consequence of conversion operations on one-hot encodings (Sect. 4.3).

For instance, to convert short encoding $\mathsf{short}(x)$ to word encoding x, we simply use the appropriate conversion on one-hot vectors, then subtract off α:

$$\mathsf{short\text{-}to\text{-}word}((\mathbf{h}, \alpha) = \mathsf{short}(\mathsf{x})) \triangleq \textbf{return } \mathsf{hot\text{-}to\text{-}word}(\mathbf{h}) - \alpha$$

This system is free (i.e., has no joins).

Conversions from a word to a short and between shorts and binary are similar:

$$\mathsf{word\text{-}to\text{-}short}(x) \triangleq \alpha \leftarrow_{\$} \mathbb{Z}_{2^k} \ ; \ \textbf{return } (\mathsf{word\text{-}to\text{-}hot}(x+\alpha), \alpha)$$
$$\mathsf{short\text{-}to\text{-}bin}((\mathbf{h}, \alpha) = \mathsf{short}(\mathsf{x})) \triangleq \textbf{return } \mathsf{hot\text{-}to\text{-}bin}(\mathbf{h}) - \alpha$$
$$\mathsf{bin\text{-}to\text{-}short}(\mathbf{x} = \mathsf{bin}(x)) \triangleq \alpha \leftarrow_{\$} \mathbb{Z}_{2^k} \ ; \ \textbf{return } (\mathsf{bin\text{-}to\text{-}hot}(\mathbf{x} + \mathsf{bin}(\alpha)), \alpha)$$

word-to-short joins $2k-1$ bits, due to the call to word-to-hot.

short-to-bin and bin-to-short requiring adding/subtracting uniform mask α *in binary representation.* Here, addition/subtraction is implemented by a ripple-carry adder built from oblivious Boolean gates (Theorem 2). It is crucial to use *oblivious* gates to hide x from E. A ripple-carry adder can be built from XORs and $k-1$ oblivious ANDs (see e.g. [24]), so the full adder joins $2k-2$ bits.

Operations on Shorts. We use our new conversions to operate on shorts. Addition/subtract/scaling by public constants are simple: convert arguments to word representation, perform the operation for free, then convert the result back to short representation. Let x, y be short encodings and let $s \in \mathbb{Z}_{2^k}$ be a constant:

$$x + y \triangleq \mathsf{word\text{-}to\text{-}short}(\mathsf{short\text{-}to\text{-}word}(x) + \mathsf{short\text{-}to\text{-}word}(y))$$
$$x - y \triangleq \mathsf{word\text{-}to\text{-}short}(\mathsf{short\text{-}to\text{-}word}(x) - \mathsf{short\text{-}to\text{-}word}(y))$$
$$s \cdot x \triangleq \mathsf{word\text{-}to\text{-}short}(s \cdot \mathsf{short\text{-}to\text{-}word}(x))$$

Each operation joins only $2k-1$ bits, due to the call to word-to-short.

Multiplication on shorts is more difficult, but can be achieved with two *half multipliers* (Sect. 4.2) and conversions. Consider two values $\mathsf{short}(x), \mathsf{short}(y)$, and suppose we wish to compute $\mathsf{short}(x \cdot y)$:

$$
\begin{array}{ll}
1 & ((\mathbf{h}_x, \alpha) = \mathsf{short}(x)) \cdot ((\mathbf{h}_y, \beta) = \mathsf{short}(y)) \triangleq \\
2 & \quad y \leftarrow \mathsf{short\text{-}to\text{-}word}(\mathsf{short}(y)) \\
3 & \quad (x + \alpha) \cdot y \leftarrow \mathsf{half\text{-}mul}(\mathbf{h}_x, y) \\
4 & \quad (y + \beta) \cdot \alpha \leftarrow \mathsf{half\text{-}mul}(\mathbf{h}_y, \alpha) \\
5 & \quad x \cdot y \leftarrow (x + \alpha) \cdot y - (y + \beta) \cdot \alpha + \alpha \cdot \beta \\
6 & \quad \textbf{return } \mathsf{word\text{-}to\text{-}short}(x \cdot y)
\end{array}
$$

Here, $\alpha \cdot \beta$ is part of the switch system's randomized input.

The full multiplication procedure joins only $4k - 1$ bits: k bits are joined per half-mul, and $2k - 1$ bits are joined as part of word-to-short.

Delayed Conversions and Inner Products. Our operations on shorts convert from words to shorts, joining $2k - 1$ bits. When handling more complex arithmetic expressions, we can as an optimization simply delay conversion back to short representation. For example, to compute the inner product of two length-n vectors of shorts, we can pointwise multiply the vector elements to obtain n words, add the words together, then perform only a single conversion back to short representation. This inner product operation thus uses $2n$ half multipliers and one word-to-short conversion, joining only $2nk + 2k - 1$ total bits.

Binary Multiplication; Modular Reduction. By using binary/short conversions with short multiplication, we can multiply k-bit binary numbers while joining only $O(k)$ bits. Because we can efficiently multiply short binary integers, we have the tools we need to apply Barrett's modular reduction algorithm (Sect. 2.5) to short integers. Namely, we can we can reduce short binary numbers modulo m while joining only $O(k)$ bits, so long as the binary number is less than m^2.

We now have all the tools we need to implement arithmetic circuits over $\mathbb{Z}_m$ for arbitrary *small* modulus $m \geq 2$. To garble an arithmetic operation over $\mathbb{Z}_m$, choose a modulus $2^k > m^2$, perform the operation on shorts in 2^k, then use Barrett reduction to simplify the result modulo m. This fact will be useful in constructing switch systems for long integers, see next.

5.2 Long Integers

All that remains is to upgrade our handling of words and shorts to handling of *long* integers, i.e. integers modulo arbitrary $m \geq 2$. Let $\ell = \lceil \log_2 m \rceil$. We show that for any choice of m, our arithmetic gates join at most $O(\ell)$ bits.

Before presenting our approach, we remark that our handling here is more theoretical than in Sect. 5.1. Our operations on longs require many conversions between representations, and this is concretely expensive. Accordingly, our presentation is less granular and less concerned with constants. Nevertheless, when garbling ℓ-bit integers, all operations join $O(\ell)$ bits. We believe our work will help lead to practically efficient garbling of long integers.

Our basic idea is to apply the Chinese Remainder Theorem (CRT, Sect. 2.4). To operate on arbitrary modulus $m \geq 2$, we consider a CRT modulus $N >$

m^2, large enough to support overflow from multiplication. We choose N as the product of the smallest distinct primes such that $N > m^2$ holds. Let r denote the number of distinct primes.

We represent a long as r shorts, where each j-th short computes operations over prime modulus p_j. We refer to each such short as a *residue*. Let P denote the largest prime modulus. Our shorts use as their word modulus the smallest modulus 2^k such that $2^k > \mathsf{max}(P^2, P \cdot \ell^2)$, sufficient to prevent overflow in all considered cases. P is at most $O(\ell)$, so $k = O(\log \ell)$; moreover, P is the $O(\ell/\log \ell)$-th prime.

Longs modulo N. To multiply/add/subtract longs modulo a CRT modulus N, we simply pairwise operate on the r residues, then reduce each j-th result modulo p_j (see discussion of modular reduction in Sect. 5.1). By the Chinese Remainder Theorem, this implements the corresponding operation modulo N. Morever, the total bit-length of all r residues is linear in $\ell = \lceil \log_2 N \rceil$, so this indeed joins only $O(\ell)$ bits per operation.

Longs modulo m; conversions to/from binary. To operate over an arbitrary modulus $\mathbb{Z}_m$, we operate over $\mathbb{Z}_N$ and simplify modulo m after each operation. To do so, we will apply Barrett's modular reduction (Sect. 2.5). However, Barrett's algorithm uses operations on *binary* integers, and our integers are currently in CRT representation. Thus, we must explain how to convert long integers to binary and vice versa. In the following, let $x \in \mathbb{Z}_N$ be the converted integer.

Converting from binary to long roughly proceeds as follows: (1) partition the bits of x into small chunks so that we can sensibly make a one-hot vector encoding each chunk, (2) use free operations on one-hot encodings to convert each chunk to CRT representation, and (3) use free addition to combine the CRT chunks into a CRT representation of x.

More precisely, we first sample a uniform mask $\alpha \in \mathbb{Z}_N$ and use oblivious Boolean gates (Theorem 2) to compute $(x+\alpha) \ominus N = [x+\alpha]_N$. Recall, $\ominus$ denotes conditional subtraction (Equation (1)), which can be achieved by a Boolean comparator and ripple-carry adder, joining only $O(\ell)$ bits.

Now that x is masked, we can reveal $[x+\alpha]_N$ to E, allowing us to use one-hot techniques. We partition the bits of $[x+\alpha]_N$ into *chunks* of $\lceil \log_2 \ell \rceil$ bits (we pad with msb zeros to ensure chunks have equal length). We use $\mathsf{bin\text{-}to\text{-}hot}$ and scale to convert each chunk to an arithmetic one-hot encoding (where one-hot vector entries are $\mathbb{Z}_{2^k}$ words).

Let $\mathcal{H}(c_i)$ be the i-th chunk, such that $x = \sum_i 2^{i \cdot \lceil \log_2 \ell \rceil} \cdot c_i$. We convert *each* chunk to r residue words. Computing each j-th residue is a free operation:

$$\langle \mathcal{T}(x \mapsto [2^{i \cdot \lceil \log_2 \ell \rceil} \cdot x]_{p_j}) \cdot \mathcal{H}(c_i) \rangle = [2^{i \cdot \lceil \log_2 \ell \rceil} \cdot c_i]_{p_j} \qquad \text{Lemma 4}$$

Now, for each chunk we have a word representing the residue of that chunk modulo each prime p_j. For each such j, we use free addition to sum all such residues. The result is a $\mathbb{Z}_{2^k}$ value *congruent to* – but not equal to – $[x+\alpha]_{p_j}$. Note that each sum is at most $(p_j - 1)$ times the number of chunks, which is $O(\ell/\log \ell)$. By our choice of k, this sum does not overflow $\mathbb{Z}_{2^k}$. To complete the conversion, we use short operations to subtract off α and reduce modulo p_j.

This full conversion joins only $O(\ell)$ bits. Its non-free operations involve (1) adding a uniform mask α, (2) converting chunks to arithmetic one-hot representation, and (3) simplification of the short residues.

Converting from long to binary leverages similar ideas. To start, G samples a uniform mask $\alpha \in \mathbb{Z}_N$ and adds $[\alpha]_{p_j}$ to each j-th residue. CRT supports the following conversion between residues $[x+\alpha]_{p_j}$ and $[x+\alpha]_N$

$$[x+\alpha]_N = \left[\sum_{i=0}^{r-1} s_j \cdot [(x+\alpha)\cdot s_j^{-1}]_{p_j}\right]_N \qquad \text{where } s_j = \frac{N}{p_j}$$

Roughly speaking, we implement this equation while using as many free operations as we can. Note that each s_j and $[s_j^{-1}]_{p_j}$ is a constant.

We use operations on shorts to compute $[(x+\alpha)\cdot s_j^{-1}]_{p_j}$ as a word. Next, we treat each constant s_j as an integer mod N, and we partition its bits into chunks of size $\lceil \log_2 \ell \rceil$. Let c_j^i denote the i-th such chunk, such that $s_j = \sum_i 2^{i\cdot\lceil\log_2 \ell\rceil}\cdot c_j^i$. We use free operations to scale the residue $[(x+\alpha)\cdot s_j^{-1}]_{p_j}$ by each chunk c_j^i. The result from each residue is $\lceil \ell/\log_2 \ell \rceil$ words, each with value at most $\lceil \log_2 \ell \rceil \cdot p_j$. These words together represent the product $s_j \cdot [(x+\alpha)\cdot s_j^{-1}]_{p_j}$. We refer to these words as the *radices* of this product.

Next, we sum each i-th radix of each residue. The result is $O(\ell/\log \ell)$ radices that jointly represent $\sum_{i=0}^{r-1} s_j \cdot [(x+\alpha)\cdot s_j^{-1}]_{p_j}$. Each sum has value at most $\lceil \ell/\log_2 \ell \rceil \cdot \ell \cdot P$. Our choice of k ensures that this does not overflow modulus 2^k.

We convert each radix to binary representation via the word-to-hot system. Note, this is oblivious due to the inclusion of uniform mask α. We use (non-oblivious) ripple-carry adders to add together all such binary integers. This may seem problematic, since we are adding $O(\ell/\log \ell)$ values, and ripple carry adders use linear gates. However, because each binary integer is only $O(\log \ell)$ bits long, we can with basic care achieve the sum while joining only $O(\ell)$ bits.

The sum of all these binary integers may be as high as $\lceil \ell/\log_2 \ell \rceil \cdot N$, and we need to simplify the result modulo N to complete the conversion. To do so, we strip the top $\lceil \log_2 \ell \rceil$ bits from the binary representation, convert these to one-hot via bin-to-hot, and use a free operation to compute a binary encoding of these bits modulo N. We then use a ripple-carry adder to recombine this with the unstripped low bits. The result is still congruent to $[x+\alpha]_N$, and now it must be lower than $2N$. Finally, we use more Boolean gates to (1) conditionally subtract off N and (2) subtract off α, resulting in a binary encoding of $[x]_N$.

Garbling Arithmetic Circuits. By combining constructions in this section, we achieve the following:

Theorem 4. *Let $m \geq 2$ be an arbitrary modulus, and let $\ell = \lceil \log_2 m \rceil$. For any n-gate arithmetic circuit C over $\mathbb{Z}_m$, there exists an oblivious switch system (S, D) that emulates C and such that* join-width$(S) = O(n \cdot \ell)$.

In the full version we prove that the system is oblivious; in short, the argument is straightforward from our inclusion of uniform masks on all values. Combining Theorem 4 with Construction 1, we achieve the following:

Corollary 1. (Arithmetic Garbled Circuits from Free XOR). *Let $m \geq 2$ be an arbitrary modulus, and let $\ell = \lceil \log_2 m \rceil$. Assuming circular correlation robust hashes (Definition 1), there exists a correct, oblivious, private, and authentic garbling scheme [7] for arithmetic circuits over $\mathbb{Z}_m$. For an n-gate circuit C, the scheme's* Garble *procedure outputs $O(n \cdot \ell \cdot \lambda)$ bits of material.*

References

1. Applebaum, B., Ishai, Y., Kushilevitz, E.: How to garble arithmetic circuits. In: Ostrovsky, R. (ed.) 52nd FOCS, pp. 120–129. IEEE Computer Society Press (2011). https://doi.org/10.1109/FOCS.2011.40
2. Ball, M., Li, H., Lin, H., Liu, T.: New ways to garble arithmetic circuits. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part II. LNCS, vol. 14005, pp. 3–34. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30617-4_1
3. Ball, M., Malkin, T., Rosulek, M.: Garbling gadgets for Boolean and arithmetic circuits. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 565–577. ACM Press (2016). https://doi.org/10.1145/2976749.2978410
4. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_24
5. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
6. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: 22nd ACM STOC, pp. 503–513. ACM Press (1990). https://doi.org/10.1145/100216.100287
7. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012, pp. 784–796. ACM Press (2012). https://doi.org/10.1145/2382196.2382279
8. Choi, S.G., Katz, J., Kumaresan, R., Zhou, H.S.: On the security of the "free-XOR" technique. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 39–53. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28914-9_3
9. Fleischhacker, N., Malavolta, G., Schröder, D.: Arithmetic garbling from bilinear maps. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019, Part II. LNCS, vol. 11736, pp. 172–192. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29962-0_9
10. Gueron, S., Lindell, Y., Nof, A., Pinkas, B.: Fast garbling of circuits under standard assumptions. J. Cryptol. **31**(3), 798–844 (2018). https://doi.org/10.1007/s00145-017-9271-y
11. Guo, C., Katz, J., Wang, X., Yu, Y.: Efficient and secure multiparty computation from fixed-key block ciphers. In: 2020 IEEE Symposium on Security and Privacy, pp. 825–841. IEEE Computer Society Press (2020). https://doi.org/10.1109/SP40000.2020.00016
12. Guo, X., et al.: Half-tree: halving the cost of tree expansion in COT and DPF. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part I. LNCS, vol. 14004, pp. 330–362. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30545-0_12

13. Harvey, D., van der Hoeven, J.: Integer multiplication in time $O(n \log n)$. Ann. Math. **193**(2), 563–617 (2021)
14. Heath, D., Kolesnikov, V.: One hot garbling. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021, pp. 574–593. ACM Press (2021). https://doi.org/10.1145/3460120.3484764
15. Heath, D., Kolesnikov, V., Ostrovsky, R.: Tri-state circuits - a circuit model that captures RAM. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 128–160. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-38551-3_5
16. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS 2010, pp. 451–462. ACM Press (2010). https://doi.org/10.1145/1866307.1866358
17. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. In: SSSR Academy of Sciences (1962)
18. Kolesnikov, V., Mohassel, P., Rosulek, M.: FleXOR: flexible garbling for XOR gates that beats free-XOR. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 440–457. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_25
19. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., et al. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
20. Lu, S., Ostrovsky, R.: How to garble RAM programs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_42
21. Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM Conference on Electronic Commerce, pp. 129–139. ACM (1999)
22. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10366-7_15
23. Rosulek, M., Roy, L.: Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part I. LNCS, vol. 12825, pp. 94–124. Springer, Heidelberg, Virtual Event (2021). https://doi.org/10.1007/978-3-030-84242-0_5
24. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: efficient MultiParty computation toolkit (2016). https://github.com/emp-toolkit
25. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS, pp. 162–167. IEEE Computer Society Press (1986). https://doi.org/10.1109/SFCS.1986.25
26. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8