Fast Garbling of Circuits over 3-Valued Logic

Yehuda Lindell and Avishay Yanai(✉)

Bar-Ilan University, Ramat Gan, Israel
yehuda.lindell@biu.ac.il, ay.yanay@gmail.com

Abstract. In the setting of secure computation, a set of parties wish to compute a joint function of their private inputs without revealing anything but the output. Garbled circuits, first introduced by Yao, are a central tool in the construction of protocols for secure two-party computation (and other tasks like secure outsourced computation), and are the fastest known method for constant-round protocols. In this paper, we initiate a study of garbling multivalent-logic circuits, which are circuits whose wires may carry values from some finite/infinite set of values (rather than only True and False). In particular, we focus on the three-valued logic system of Kleene, in which the admissible values are True, False, and Unknown. This logic system is used in practice in SQL where some of the values may be missing. Thus, efficient constant-round secure computation of SQL over a distributed database requires the ability to efficiently garble circuits over 3-valued logic. However, as we show, the two natural (naive) methods of garbling 3-valued logic are very expensive.

In this paper, we present a general approach for garbling three-valued logic, which is based on first encoding the 3-value logic into Boolean logic, then using standard garbling techniques, and final decoding back into 3-value logic. Interestingly, we find that the specific encoding chosen can have a significant impact on efficiency. Accordingly, the aim is to find Boolean encodings of 3-value logic that enable efficient Boolean garbling (i.e., minimize the number of AND gates). We also show that Boolean AND gates can be garbled at the same cost of garbling XOR gates in the 3-value logic setting. Thus, it is unlikely that an analogue of free-XOR exists for 3-value logic garbling (since this would imply free-AND in the Boolean setting).

1 Introduction

1.1 Background – Three-Valued Logic

In classical (Boolean) propositional logic, statements are assigned a "truth-value" that can be either True or False, but not both. Logical operators are

Y. Lindell and A. Yanai—Supported by the European Research Council under the ERC consolidators grant agreement no. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

M. Abdalla and R. Dahab (Eds.): PKC 2018, LNCS 10769, pp. 620–643, 2018.
https://doi.org/10.1007/978-3-319-76578-5_21

used to make up a complex statement out of other, one or more, simpler statements such that the truth value of the complex statement is derived from the simpler ones and the logical operators that connects them. For instance, given that the statement A is True and the statement B is True we infer that the statement $C =$"A and B" (denoted by $C = A \wedge B$) is True as well.

Another branch of propositional logic is the *multivalent logic system*. Multivalent logic systems consider more than two truth-values, that is, they may admit anything from three to an infinite number of possible truth-values. Among those, the simplest and most studied sort is the *three-valued logic* (or ternary logic), which is a system that admits three truth-values, e.g., "truth", "falsity" and "indeterminancy". Such a system seems to suit many real life situations, for instance, statements about the future or paradoxical statements like *"this statement is not correct"*, which must have an indeterminate truth-value. Note that in different applications, the third truth-value could be interpreted differently, hence, different inference rules are derived[1]. The most common three-valued logic system is Kleene's Logic [6], in which statements are assigned with either True, False or Unknown. For clarity, whenever we use the term three-valued logic or 3VL we actually refer to Kleene's Logic. We remark that although other three-valued logic system exist, in this paper we focus only on Kleene's logic since its use in real life is the most prevalent; see the application example in Sect. 1.2.

The admission of Unknown requires one to expand the set of inference rules, to enable the computation of the truth-value of a complex statement from simpler statements, even if one or more of them are Unknown. In Kleene's logic, the inference process complies with the way we usually make conclusions: It yields Unknown whenever at least one statement that is necessary for deciding True or False is assigned with Unknown. For example, the AND of True and Unknown is Unknown since if the Unknown were False then the result would be false. However, the OR of True and Unknown is True since it equals True irrespective of the Unknown variable's value.

The 3VL inference rules are presented in Table 1 in the form of truth tables. In the rest of the paper whenever we refer to the Boolean version of AND, OR,

Table 1. Definition of the functions $\wedge_3$ (AND), $\vee_3$ (OR), $\oplus_3$ (XOR) and $\neg_3$ (NOT) using truth tables. Note these functions are symmetric, that is, the order of the inputs makes no difference.

$\wedge_3$	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

$\vee_3$	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

$\oplus_3$	T	U	F
T	F	U	T
U	U	U	U
F	T	U	F

	$\neg_3$
T	F
U	U
F	T

[1] In fact, even the two traditional truth-values True and False could have other meaning in different three-valued logic systems.

XOR and NOT we use the usual notation $\wedge, \vee, \oplus, \neg$ and when we use their 3VL version we subscript it with the number 3, e.g. $\wedge_3, \vee_3, \oplus_3, \neg_3$. We denote by T, F and U, the 3VL values True, False and Unknown, respectively.

1.2 Applications in SQL

In SQL specifications [5] the NULL marker indicates the absence of a value, or alternatively, that the value is neither True nor False, but Unknown. Because of this, comparisons with NULL never result in either True or False, but always in the third logical value: Unknown. For example, the statement "SELECT 10 = NULL" results in Unknown. However, certain operations on Unknown can return values if the absent value is not relevant to the outcome of the operation. Consider the following example:

```
SELECT * FROM T1 WHERE (age > 30 OR height < 140) AND weight > 110
```

Now, consider an entry where the person's age is missing. In this case, if the person's height is 150 then the OR subexpression evaluates to Unknown and so the entire result is Unknown, hence, this entry is not retrieved. In contrast, if the person's height is 120, then the OR subexpression evaluates to True, and so the result is True if weight $>$ 110, and False if weight $\leq$ 110.

We remark that the main SQL implementations [2,9,10] (Oracle, Microsoft and MySQL) conform to the Kleene's three-valued logic described above. As such, if secure computation is to be used to carry out secure SQL on distributed (or shared) databases, then efficient solutions for dealing with three-valued logic need to be developed.

1.3 Naively Garbling a 3VL Gate

We begin by describing the straightforward (naive) approach to garbling a 3VL gate. Let g_3 be a 3VL gate with input wires x, y and output wire z, where each wire takes one of 3 values, denoted T, F and U. The basic garbling scheme of Yao [8,13] works by associating a random key with each possible value on each wire, and then encrypting each possible output value under all combinations of input values that map to that output value. Specifically, for each wire $\alpha \in \{x, y, z\}$, choose random keys $k_\alpha^T, k_\alpha^F, k_\alpha^U$. Then, for every combination of $\beta_x, \beta_y \in \{T, F, U\}$, encrypt $k_z^{g(\beta_x,\beta_y)}$ using keys $k_x^{\beta_x}, k_y^{\beta_y}$ and define the garbled table to be a random permutation of the ciphertexts. See Fig. 1 for a definition of such a garbled gate.[2]

[2] Note that in a two-party protocol like Yao's, the parties then run 1-out-of-3 oblivious transfers in order for the evaluator to learn the keys that are associated with its input.

This approach yields a garbled gate of 9 entries. Using the standard garbled row reduction technique [11], it is possible to reduce the size of the gate to 8 entries. This means that 8 ciphertexts need to be communicated for each gate in the circuit. However, this garbling scheme requires *four times* more bandwidth for three-valued logic gates than the state-of-the-art for their Boolean $\wedge$ counterparts [12]. Furthermore, using the free-XOR paradigm [7] (as is also utilized in [12]), XOR gates are free in the Boolean case but require significant bandwidth and computation in the three-valued logic case. (We remark that [7,12] do require non-standard assumptions; however, these techniques do not translate to the 3VL case and so cannot be used, even under these assumptions.)

1	$E_{k_x^T}\left(E_{k_y^T}\left(k_z^{g(T,T)}\right)\right)$
2	$E_{k_x^T}\left(E_{k_y^F}\left(k_z^{g(T,F)}\right)\right)$
3	$E_{k_x^T}\left(E_{k_y^U}\left(k_z^{g(T,U)}\right)\right)$
4	$E_{k_x^F}\left(E_{k_y^T}\left(k_z^{g(F,T)}\right)\right)$
5	$E_{k_x^F}\left(E_{k_y^F}\left(k_z^{g(F,F)}\right)\right)$
6	$E_{k_x^F}\left(E_{k_y^U}\left(k_z^{g(F,U)}\right)\right)$
7	$E_{k_x^U}\left(E_{k_y^T}\left(k_z^{g(U,T)}\right)\right)$
8	$E_{k_x^U}\left(E_{k_y^F}\left(k_z^{g(U,F)}\right)\right)$
9	$E_{k_x^U}\left(E_{k_y^U}\left(k_z^{g(U,U)}\right)\right)$

Fig. 1. Garbling a 3VL gate directly using 9 rows.

Before proceeding, we note that another natural way of working is to translate each variable in the 3VL circuit into two Boolean variables: the first variable takes values T, F (true/false), and the second variable takes values K, U (known/unknown). This method fits into our general paradigm for solving the problem and so will be described later; as we will show, this specific method is not very efficient.

1.4 Our Results

The aim of this paper is to find ways of garbling three-valued logic functions that are significantly more efficient than the naive method described in Sect. 1.3. Our methods all involve first encoding a 3VL function as a Boolean function and then utilizing the state-of-the-art garbling schemes for Boolean functions. These schemes have the property that AND gates are garbled using two ciphertexts, and XOR gates are garbled for free [7,12]. Thus, our aim is to find Boolean encodings of 3VL functions that can be computed using few AND gates (and potentially many XOR gates).

In order to achieve our aim, we begin by formalizing the notion of a 3VL-Boolean encoding which includes a way of encoding 3VL-input into Boolean values, and a way of computing the 3VL function using a Boolean circuit applied to the encoded input. Such an encoding reduces the problem of evaluating 3VL functions to the problem of evaluating Boolean functions. Our formalization is general, and can be used to model other multivalent logic systems, like that used in fuzzy logic.

Next, we construct *efficient* 3VL-Boolean encodings, where by efficient, we mean encodings that can be computed using few Boolean AND gates. Interestingly, we show that the way that 3VL-variables are encoded as Boolean variables has a great influence on the efficiency of the Boolean computation. We describe three different encodings: The first encoding is the natural one, and it works by defining two Boolean variables x_T and x_U for every 3VL-variable x such that $x_U = 1$ if and only if $x = U$, and $x_T = 1$ if $x = T$ and $x_T = 0$ if $x = F$. This is "natural" in the sense that one Boolean variable is used to indicate whether the 3VL-value is known or not, and the other variable is used to indicate whether the 3VL-value is true or false in the case that it is known. We show that under this encoding, 3VL-AND gates can be computed at the cost of 6 Boolean AND gates, and 3VL-XOR gates can be computed at the cost of 1 Boolean AND gate. We then proceed to present two alternative encodings; the first achieves a cost of 4 Boolean AND gates for every 3VL-AND gate and 1 Boolean AND gate for every 3VL-XOR gate, whereas the second achieves a cost of 2 Boolean AND gates both for every 3VL-AND gate and every 3VL-XOR gate. These encodings differ in their cost tradeoff, and the choice of which to use depends on the number of AND gates and XOR gates in the 3VL-circuit.

Given these encodings, we show how *any* protocol for securely computing Boolean circuits, for semi-honest or malicious adversaries, can be used to securely compute 3VL circuits, at almost the same cost. Our construction is black-box in the underlying protocol, and is very simple.

Finally, observe that all our encodings have the property that 3VL-XOR gates are computed using at least 1 Boolean AND-gate. This means that none of our encodings enjoy the free-XOR optimization [7] which is extremely important in practice. We show that this is actually somewhat inherent. In particular, we show that it is possible to garble a Boolean AND gate at the same cost of garbling a 3VL XOR gate. Thus, free-3VL-XOR would imply free-Boolean-AND, which would be a breakthrough for Boolean garbling. Formally, we show that free-3VL-XOR is *impossible* in the linear garbling model of [12].

Brute-force search for encodings. It is theoretically possible to search for efficient 3VL-Boolean encodings by simply trying all functions with a small number of AND gates, for every possible encoding. Indeed, for up to *one* AND gate it is possible since the search space is approximately 2^{20} possibilites. However, if up to *two* AND gates are allowed, then the search space already exceeds 2^{50} possibilities. We ran a brute-force search for up to one AND gate, and rediscovered our 3VL-XOR computation that uses a single AND gate (in fact, we found multiple ways of doing this). However, our search showed that there does *not* exist a way of computing 3VL-AND using a single AND gate, for *any encoding*. See Appendix A for more details on the brute-force search algorithm that we used.

2 Encoding 3VL Functions as Boolean Functions

2.1 Notation

We denote by T, V, U the 3VL values $\mathsf{True}, \mathsf{False}$ and $\mathsf{Unknown}$, respectively, and by $1, 0$ the Boolean values True and False. We denote by F_3 the set of all 3VL functions (i.e. all functions of the form $\{T, F, U\}^* \rightarrow \{T, F, U\}^*$) and by F_2 be the set of all Boolean functions (i.e. all functions of the form $\{0, 1\}^* \rightarrow \{0, 1\}^*$).

In addition, we denote by $\mathsf{F}_3(\ell, m)$ and $\mathsf{F}_2(\ell, m)$ the set of all 3VL and Boolean functions, respectively, that are given ℓ inputs and produce m outputs. We denote by x_i the ith element in x both for $x \in \{T, F, U\}^*$ and $x \in \{0, 1\}^*$.

2.2 3VL-Boolean Encoding

As we have mentioned, in order to utilize the efficiency of modern garbling techniques, we reduce the problem of garbling 3VL circuits to the problem of garbling Boolean circuits, by encoding 3VL functions as Boolean functions. Informally speaking, a 3VL-Boolean encoding is a way of mapping 3VL inputs into Boolean inputs, computing a Boolean function on the mapped inputs, and mapping the Boolean outputs back to a 3VL output. This method is depicted in Fig. 2. The naive approach appears on the left and involves directly garbling a 3VL circuit, as described in Sect. 1.3. Our approach appears on the right and works by applying a transformation $\mathsf{Tr}_{3\to 2}$ to map the 3VL input to a Boolean input, then computing an appropriately defined Boolean function, and finally applying a transformation $\mathsf{Tr}_{2\to 3}$ to map the output back. The Boolean function is also defined by a transformation, so that a 3VL function f_3 is transformed to a Boolean function f_2 via the transformation Tr_F, that is, $f_2 = \mathsf{Tr}_F(f_3)$, and this is what is computed. As such, as we will see, it suffices to garble the Boolean function f_2, and if this function has few AND gates then it will be efficient for this purpose.

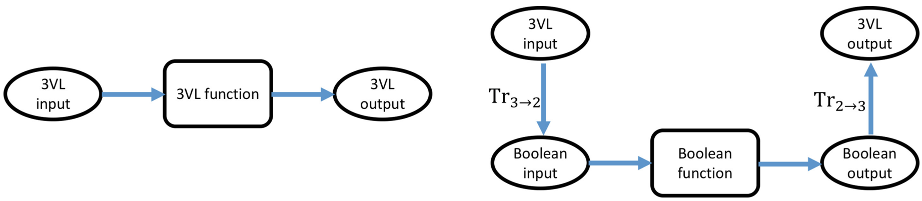

Fig. 2. Naive approach on the left side and our new approach on the right side.

Observe that since we map inputs from three-valued logic to Boolean logic, the set sizes of all possible inputs are different. Thus, we define encodings via *relations* and not via bijective mappings. Of course, the actual transformations $\mathsf{Tr}_{3\to 2}$ and $\mathsf{Tr}_{2\to 3}$ are functions. However, the mapping between inputs and outputs may be expressed as relations; e.g., when mapping a single 3VL variable to two Boolean variables, it may be the case that one of the 3VL variables can be expressed as two possible Boolean pairs. This enables more generality, and can help in computation, as we will see below.

Although one could define a very general encoding from 3VL to Boolean values, we will specifically consider encodings that map every single 3VL variable to exactly two Boolean variables. We consider this specific case since it simplifies our definitions, and all our encodings have this property.

The formal definition. Let $f : \{T, F, U\}^m \rightarrow \{T, F, U\}^n$ be a 3VL function. We begin by defining the appropriate relations and transformations.

1. A value encoding is a relation $R_{3\rightarrow 2} \subseteq \{T, F, U\} \times \{0, 1\}^2$ that is left-total and injective.[3] For $\ell \in \mathbb{N}$, let $R^{\ell}_{3\rightarrow 2} \subseteq \{T, F, U\}^{\ell} \times \{0, 1\}^{2\ell}$ be the relation defined by extending $R_{3\rightarrow 2}$ per coordinate.[4]
2. A valid input transformation is an injective function $\mathsf{Tr}^m_{3\rightarrow 2} : \{T, F, U\}^m \rightarrow \{0, 1\}^{2m}$ such that $\mathsf{Tr}^m_{3\rightarrow 2} \subseteq R^m_{3\rightarrow 2}$. Note that since $R_{3\rightarrow 2}$ is a relation, there may be multiple different input transformations.
3. A function transformation $\mathsf{Tr}^{m,n}_F : \mathsf{F}_3(m, n) \rightarrow \mathsf{F}_2(2m, 2n)$ is a function that converts 3VL functions to Boolean functions with appropriate input-output lengths.
4. The output transformation $\mathsf{Tr}^n_{2\rightarrow 3} : \{0, 1\}^{2n} \rightarrow \{T, F, U\}^n$ is the inverse of $R_{3\rightarrow 2}$. That is, $\mathsf{Tr}^1_{2\rightarrow 3}((b_1, b_2)) = x$ for every $(x, (b_1, b_2)) \in R_{3\rightarrow 2}$. Note that since $R_{3\rightarrow 2}$ is injective, this transformation is *unique*.

Observe that $R_{3\rightarrow 2}$ is required to be injective since otherwise a Boolean value y could represent two possible 3VL values x, z, and so the output cannot be uniquely mapped back from a Boolean value to a 3VL value. Furthermore, note that by requiring $\mathsf{Tr}^m_{3\rightarrow 2} \subseteq R^m_{3\rightarrow 2}$, we have that the transformation constitutes a valid encoding according to the relation.

Informally, a 3VL-Boolean encoding is such that the process of transforming the inputs, computing the transformed Boolean function, and transforming the outputs back, correctly computes the 3VL function. Our definition of an encoding includes the value encoding and function transformation only, and we require that it works correctly for *all* input transformations; we discuss why this is the case below.

Definition 2.1. *Let $m, n \in \mathbb{N}$; let $R_{3\rightarrow 2}$ be a value encoding, and let $\mathsf{Tr}^{m,n}_F$ be a function transformation. Then, the pair $(R^m_{3\rightarrow 2}, \mathsf{Tr}^{m,n}_F)$ is a* 3VL-Boolean Encoding of $\mathsf{F}_3(m, n)$ *if for every $f_3 \in \mathsf{F}_3(m, n)$, every valid input transformation $\mathsf{Tr}^m_{3\rightarrow 2}$, and every $x \in \{T, F, U\}^m$:*

$$\mathsf{Tr}^n_{2\rightarrow 3}\big(f_2\big(\mathsf{Tr}^m_{3\rightarrow 2}(x)\big)\big) = f_3(x) \tag{1}$$

where $f_2 = \mathsf{Tr}^{m,n}_F(f_3)$.

The above definition simply states that computing via the transformations yields correct output. However, as we have mentioned, we require that this works for *all* input transformations and not just for a specific one. It may seem more natural to define a 3VL-Boolean encoding in which the input transformation $\mathsf{Tr}^m_{3\rightarrow 2}$ is fixed, rather than requiring that Eq. (1) holds for *every* valid input

[3] A relation R from X to Y is left-total if for all $x \in X$ there exists $y \in Y$ such that $(x, y) \in R$. R is injective if for every $x, z \in X$ and $y \in Y$, if $(x, y) \in R$ and $(z, y) \in R$ then $x = z$.

[4] That is, $\big((A_1, \ldots, A_\ell), ((b_1, b_2), \ldots, (b_{2\ell-1}, b_{2\ell}))\big) \in R^{\ell}_{3\rightarrow 2}$ if and only if for every $1 \leq i \leq \ell$ it holds that $(A_i, (b_{2i-1}, b_{2i})) \in R_{3\rightarrow 2}$.

transformation. However, in actuality, it is quite natural to require that the transformed function work for every input transformation since this means that it works for every possible mapping of three-valued inputs to their Boolean counterparts. More significantly, this property is *essential* for proving the composition theorem of Sect. 2.3 that enables us to compose different function encodings together. As we will see, this is important since it enables us to define independent encodings for different types of gates, and then compose them together to compute any function.

2.3 Composition of 3VL Functions

In this section, we prove that encodings can be composed together. Specifically, we prove that for any two 3VL functions g_3 and f_3 and any 3VL input x, computing $g \circ f(x)$ yields the same value as when g, f, x are *separately* transformed into g', f', x' using *any valid* 3VL-Boolean encoding, and then the output of $g' \circ f'(x')$ is transformed back to its 3VL representation. As we will see, this is very important since it enables us to define independent encodings on different types of gates, and then compose them together to compute any function. Formally:

Theorem 2.2. *Let m, ℓ, n be natural numbers, and let $R_{3\to2}$ be a value encoding. Let $\mathsf{E}_1 = \left(R_{3\to2}, \mathsf{Tr}_F^{m,\ell}\right)$ and $\mathsf{E}_2 = \left(R_{3\to2}, \mathsf{Tr}_F^{\ell,n}\right)$ be two 3VL-Boolean encodings (with the same relation $R_{3\to2}$). Then, for every $f_3 \in \mathsf{F}_3(m, \ell)$, every $g_3 \in \mathsf{F}_3(\ell, n)$, every input transformation $\mathsf{Tr}_{3\to2}^m$, and every $x \in \{T, F, U\}^m$:*

$$\mathsf{Tr}_{2\to3}^n\left(g_2\left(f_2\left(\mathsf{Tr}_{3\to2}^m(x)\right)\right)\right) = g_3(f_3(x)),$$

where $f_2 = \mathsf{Tr}_F^{m,\ell}(f_3)$ and $g_2 = \mathsf{Tr}_F^{\ell,n}(g_3)$. Equivalently, $\left(R_{3\to2}, \mathsf{Tr}_F^{\ell,n} \circ \mathsf{Tr}_F^{m,\ell}\right)$ is a 3VL-Boolean encoding of $\mathsf{F}_3(m, n)$.

Before proving the theorem, we present the following claim which simply express that if the output transformation of an encoding maps a Boolean value $\tilde{Y}$ to some 3VL value y then there must exist a input transformation that maps y to $\tilde{Y}$. Formally:

Claim 2.1. *Let $R_{3\to2}$ be a valid value encoding and let $\mathsf{Tr}_{3\to2}, \mathsf{Tr}_{2\to3}$ be a valid input and output transformations respectively such that for $\tilde{Y} \in \{0,1\}^2$ it holds that $\mathsf{Tr}_{2\to3}(\tilde{Y}) = y$ and $\mathsf{Tr}_{3\to2}(y) = Y$. Then there exists a valid input transformation $\tilde{\mathsf{Tr}}_{3\to2}$ (with respect to $R_{3\to2}$) such that $\tilde{\mathsf{Tr}}_{3\to2}(y) = \tilde{Y}$.*

Proof: If $Y = \tilde{Y}$ then there is nothing to prove, i.e. $\tilde{\mathsf{Tr}}_{3\to2} = \mathsf{Tr}_{3\to2}$. Consider the case of $Y \neq \tilde{Y}$: This means that $R_{3\to2}$ maps the 3VL value y to both Boolean pairs Y and $\tilde{Y}$. Denote the other two 3VL values by y' and y'' and similarly the remaining Boolean pairs by Y' and Y'' such that $R_{3\to2}(y') = Y'$. It is immediate that the two valid transformations (with respect to $R_{3\to2}$) are

$$\begin{aligned}\mathsf{Tr}_{3\to2} &= \{y' \mapsto Y', y'' \mapsto Y'', y \mapsto Y\} \quad \text{and} \\ \tilde{\mathsf{Tr}}_{3\to2} &= \left\{y' \mapsto Y', y'' \mapsto Y'', y \mapsto \tilde{Y}\right\}\end{aligned}$$

■

Proof: [of Theorem 2.2]**.** By the validity of encodings $\mathsf{E}_1, \mathsf{E}_2$ (Definition 2.1) it follows that for value encoding $R_{3\to2}$ and every valid input transformations $\mathsf{Tr}^{\ell}_{3\to2}, \mathsf{Tr}^{m}_{3\to2}$, every $f_3 \in \mathsf{F}_3(m,\ell)$, $g_3 \in \mathsf{F}_3(\ell,n)$ and every $x \in \{T,F,U\}^m$:

$$g_3\left(f_3\left(x\right)\right) = \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\mathsf{Tr}^{\ell}_{3\to2}\left(\mathsf{Tr}^{\ell}_{2\to3}\left(f_2\left(\mathsf{Tr}^{m}_{3\to2}\left(x\right)\right)\right)\right)\right)\right) \tag{2}$$

where $f_2 = \mathsf{Tr}^{m,\ell}_{F}(f_3)$ and $g_2 = \mathsf{Tr}^{\ell,n}_{F}(g_3)$. This is true due to the following: Let $y_f = \mathsf{Tr}^{\ell}_{2\to3}\left(f_2\left(\mathsf{Tr}^{m}_{3\to2}\left(x\right)\right)\right)$. By Definition 2.1 y_f is guaranteed to be equal to $f_3(x)$ and $y_g = \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\mathsf{Tr}^{\ell}_{3\to2}\left(y_f\right)\right)\right)$ is guaranteed to be equal to $g_3(y_f)$. Concluding that the right hand-side of Eq. 2 equals $g_3(f_3(x))$. In the following we show that we can remove the two intermediate transformations $\mathsf{Tr}^{\ell}_{3\to2}, \mathsf{Tr}^{\ell}_{2\to3}$ from the Eq. (2) and obtain the same result: Let

$$\begin{aligned}Y &= f_2(\mathsf{Tr}^{m}_{3\to2}(x)) \quad \text{and} \\ \hat{y} &= \mathsf{Tr}^{\ell}_{2\to3}(Y)\end{aligned}$$

Let $\hat{\mathsf{Tr}}^{\ell}_{3\to2}$ be a valid input transformation (with respect to $R_{3\to2}$) such that $\hat{\mathsf{Tr}}^{\ell}_{3\to2}(\hat{y}) = Y$ (there must exist such a transformation from Claim 2.1). We get:

$$\begin{aligned}g_3(f_3(x)) &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\mathsf{Tr}^{\ell}_{3\to2}\left(\mathsf{Tr}^{\ell}_{2\to3}\left(f_2\left(\mathsf{Tr}^{m}_{3\to2}\left(x\right)\right)\right)\right)\right)\right) \\ &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\mathsf{Tr}^{\ell}_{3\to2}\left(\mathsf{Tr}^{\ell}_{2\to3}(Y)\right)\right)\right) \\ &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\mathsf{Tr}^{\ell}_{3\to2}\left(\hat{y}\right)\right)\right) \\ &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(\hat{\mathsf{Tr}}^{\ell}_{3\to2}\left(\hat{y}\right)\right)\right) \\ &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(Y\right)\right) \\ &= \mathsf{Tr}^{n}_{2\to3}\left(g_2\left(f_2(\mathsf{Tr}^{m}_{3\to2}(x))\right)\right)\end{aligned}$$

as required. The 2nd equation follows from the definition of Y; the 3rd follows from definition of $\hat{y}$; the 4th follows from the fact that E_2 is a valid encoding and must work for every valid input transformation, in particular it must work with $\hat{\mathsf{Tr}}^{\ell}_{3\to2}$; the 5th follows from the way we chose $\hat{\mathsf{Tr}}^{\ell}_{3\to2}$, i.e. $\hat{\mathsf{Tr}}^{\ell}_{3\to2}(\hat{y}) = Y$ and the 6th equation follows from the definition of Y. Concluding the correctness of the theorem. ■

We remark that it is crucial that the two encodings in Theorem 2.2 be over the *same* relation and that the encodings be such that they work for all input transformations (as required in Definition 2.1). In order to see why, consider for a moment what could happen if the definition of an encoding considered a *specific* input transformation and was only guaranteed to work for this transformation. Next, assume that f_2 outputs a Boolean pair that is not in the range of the input transformation specified for E_2. In this case, g_2 may not work correctly and so the composition may fail. We remark that this exact case occurred in natural constructions that we considered in the process of this research. This explains why correctness is required for all possible input transformations in Definition 2.1.

Using Theorem 2.2. This composition theorem is important since it means that we can construct separate encodings for each gate type and then these can be combined in the natural way. That is, it suffices to separately find (efficient) *function transformations* for the functions $\wedge_3, \neg_3$ and $\oplus_3$ and then every 3VL function can be computed by combining the Boolean transformations of these gates. Note that since $\neg_3$ is typically free and De Morgan's law holds for this three-valued logic as well, we do not need to separately transform $\vee_3$.

2.4 More Generalized Encodings

In order to simplify notation, we have defined encodings to be of the form that every 3VL value $x \in \{T, F, U\}$ is mapped to a pair of Boolean bits; indeed, all of our encodings in this paper are of this form. However, we stress that our formalization is general enough to allow other approaches as well. In particular, it is possible to generalize our definition to allow more general encodings from $x \in \{T, F, U\}^m$ to $y \in \{0,1\}^\ell$ that could result in $\ell < 2m$. In addition, it is conceivable that mapping $x \in \{T, F, U\}$ to *more* than 2 bits may yield more efficient function transformations with respect to the number of Boolean gates required to compute them. These and other possible encodings can easily be captured by a straightforward generalization of our definition.

3 A Natural 3VL-Boolean Encoding

In this section we present our first 3VL-Boolean encoding which we call the "natural" encoding. This encoding is natural in the sense that a 3VL value x is simply transformed to a pair of 2 Boolean values (x_U, x_T), such that x_U signals whether the value is known or unknown, and x_T signals whether the value is true or false (assuming that it is known). Formally, we define

$$R_{3\to 2} = \big\{ \big(T, (0,1)\big),\ \big(F, (0,0)\big),\ \big(U, (1,0)\big),\ \big(U, (1,1)\big) \big\},$$

and so U is associated with two possible values $(1,0), (1,1)$, and T and F are each associated with a single value. Note that $R_{3\to 2}$ is left-total and injective as required by the definition. We now define the input transformation function $\mathsf{Tr}_{3\to2}$ which is defined by mapping T and F to their unique images, and maps U to one of $(1,0)$ or $(0,1)$. For concreteness, we define:

$$(x_U, x_T) = \mathsf{Tr}_{3\to2}(x) = \begin{cases} (0,1) & x = T \\ (0,0) & x = F \\ (1,0) & x = U \end{cases}.$$

We proceed to define the function transformation Tr_F. As we have discussed, it is possible to define many such transformations, and our aim is to find an "efficient one" with as few AND gates as possible. Below, we present the most efficient transformations of $\wedge_3, \oplus_3, \neg_3$ that we have found for this encoding. As mentioned in Sect. 2.3, these suffice for computing any function (and $\vee_3$ can be computed using $\wedge_3$ and $\neg_3$ by De Morgan's law).

Let $x, y \in \{T, F, U\}$ be the input values, and let z be the output value. We denote $\mathsf{Tr}_{3\to2}(x) = (x_U, x_T)$ meaning that (x_U, x_T) is the Boolean encoding of x; likewise for y and z. We define the transformations below. All of these transformations work by computing z_T as the standard logical operation over the x_T, y_T variables (since these indicate T/F), and compute the z_U based on the reasoning as to when the output is unknown. We have:

1. $\mathsf{Tr}_F(\wedge_3)$ outputs the function $\wedge_2(x_U, x_T, y_U, y_T) = (z_U, z_T)$, defined by

   $$z_U = (x_U \wedge y_U) \vee (x_U \wedge y_T) \vee (x_T \wedge y_U) \quad \text{and} \quad z_T = x_T \wedge y_T.$$

   As mentioned above, $z_T = x_T \wedge y_T$ which gives the correct result and will determine the value if it is known. Regarding z_U, observe that $z_U = 1$ if both x and y equal U or if one of them is U and the other is T (which are the exact cases that the result is unknown). Furthermore, if either of x or y equals F (and so the result should be known), then $z_U = 0$, as required.
2. $\mathsf{Tr}_F(\oplus_3)$ outputs the function $\oplus_2(x_U, x_T, y_U, y_T) = (z_U, z_T)$, defined by

   $$z_U = x_U \vee y_U \quad \text{and} \quad z_T = x_T \oplus y_T.$$

   Once again, $z_T = x_T \oplus y_T$ which is correct if the value is known. Regarding z_U, recall that for XOR, if either input is unknown then the result is unknown. Thus, $z_U = x_U \vee y_U$.
3. $\mathsf{Tr}_F(\neg_3)$ outputs the function $\neg_2(x_U, x_T) = (z_U, z_T)$, defined by

   $$z_U = x_U \quad \text{and} \quad z_T = \neg x_T,$$

   which is correct since z_T is computed as above, and $z_U = x_U$ since the output of a negation gate is unknown if and only if the input is unknown.

Correctness. The formal proof that this is a valid encoding is demonstrated simply via the truth tables of each encoding. This can be found in Appendix C.1.

Efficiency. The transformations above have the following cost: $\wedge_3$ can be computed at the cost of 6 Boolean $\wedge$ and $\vee$ gates (5 for computing z_U and one for computing z_T), $\oplus_3$ can be computed at the cost of a single Boolean $\vee$ and a single Boolean $\oplus$ gate, and $\neg_3$ can be computed at the cost of a single Boolean $\neg$ gate. (We ignore $\neg$ gates from here on since they are free in all known garbling schemes.)

Concretely, when using the garbling scheme of [12] that incorporates free-XOR and requires two ciphertexts for $\vee$ and $\wedge$, we have that the cost of garbling $\vee_3$ is 12 ciphertexts, and the cost of garbling $\oplus_3$ is 2 ciphertexts. In comparison, recall that the naive garbling scheme of Sect. 1.3 required 8 ciphertexts for both $\vee_3$ and $\oplus_3$. In order to see which is better, let C be a 3VL circuit and denote by $C_\wedge$ and $C_\oplus$ the number of $\wedge_3$ and $\oplus_3$ gates in C, respectively. Then, the natural 3VL-Boolean encoding is better than the naive approach of Sect. 1.3 if and only if

$$12 \cdot C_\wedge + 2 \cdot C_\oplus < 8 \cdot C_\wedge + 8 \cdot C_\oplus,$$

which holds if and only if $C_\wedge < 1.5 \cdot C_\oplus$. This provides a clear tradeoff between the methods. We now proceed to present encodings that are strictly more efficient than both the natural 3VL Boolean encoding and the naive garbling of Sect. 1.3.

4 A More Efficient Encoding Using a Functional Relation

In this section we present a 3VL-Boolean encoding, in which the relation $R_{3\to 2}$ is *functional*.[5] Since $R_{3\to 2}$ is already left-total and injective, this implies that $R_{3\to 2}$ is in fact a 1-1 function. We define $R_{3\to 2} = \{(T,(1,1)),\ (F,(0,0)),\ (U,(1,0))\}$. Since $R_{3\to 2}$ is a 1-1 function, there is only one possible input transformation $(x_T, x_F) = \mathsf{Tr}_{3\to 2} = R^{-1}_{3\to 2}$. The intuition behind this encoding is as follows: The value $x \in \{T, F, U\}$ is mapped to a pair (x_T, x_F) so that if x is true or false then $x_T = x_F$, appropriately (i.e., if $x = T$ then $x_T = x_F = 1$, and if $x = F$ then $x_T = x_F = 0$). In contrast, if x is unknown, then x_T and x_F take different values of 1 and 0, respectively, representing an "unknown" state (both 1 and 0). We denote the Boolean values x_T and x_F because in case that $x = U$ then x_T is assigned with True and x_F is assigned with False.

As we will see, it is possible to compute $\wedge_3$, $\oplus_3$ and $\neg_3$ gates under this encoding at a cost that is strictly more efficient than the natural encoding of Sect. 3. In order to show this, in Sect. 4.1, we begin by presenting a simple transformation Tr_F for $\wedge_3$ and $\neg_3$ gates. These are clearly complete, and furthermore are the most common connectives used in the context of SQL (as above, $\neg_3$ is "free" and so $\vee_3$ can be transformed at the same cost as $\wedge_3$). However, for the general case, an efficient transformation for $\oplus_3$ gates is also desired since the naive method of computing $\oplus$ from $\wedge, \vee, \neg$ is quite expensive. We therefore show how to also deal with $\oplus_3$ gates in Sect. 4.2.

[5] Relation R from X to Y is functional if for all $x \in X$ and $y, z \in Y$ it holds that if $(x, y) \in R$ and $(x, z) \in R$ then $y = z$. Stated differently, R is a function.

4.1 An Efficient Function Transformation for $\wedge_3, \neg_3$ Gates

We now show how to transform $\wedge_3$ and $\neg_3$ gates into Boolean forms at a very low cost: $\wedge_3$ gates can be transformed at the cost of just two Boolean $\wedge$ gates, and $\neg_3$ gates can be transformed at the cost of two Boolean $\neg$ gates (which are free in all garbling schemes).

1. $\mathsf{Tr}_F(\wedge_3)$ outputs the function $\wedge_2(x_T, x_F, y_T, y_F) = (z_T, z_F)$, defined by
$$z_T = x_T \wedge y_T \quad \text{and} \quad z_F = x_F \wedge y_F.$$
2. $\mathsf{Tr}_F(\neg_3)$ outputs the function $\neg_2(x_T, x_F) = (z_T, z_F)$, defined by
$$z_T = \neg x_F \quad \text{and} \quad z_F = \neg x_T.$$

We now prove that these transformations are correct. We begin with $\mathsf{Tr}_F(\wedge_3)$:

1. If $x \wedge y = T$ then $x = y = T$ and so $x_T = x_F = y_T = y_F = 1$. Thus, $z_T = z_F = 1$ which means that $z = \mathsf{Tr}_{2\to3}(z_T, z_F) = \mathsf{Tr}_{2\to3}(1,1) = T$, as required.
2. If $x \wedge y = F$, then either $x = F$ which means that $x_T = x_F = 0$, or $y = F$ which means that $y_T = y_F = 0$, or both. This implies that $z_T = z_F = 0$ and so $z = \mathsf{Tr}_{2\to3}(z_T, z_F) = \mathsf{Tr}_{2\to3}(0,0) = F$, as required.
3. Finally, if $x \wedge y = U$, then we have three possible cases:
   (a) *Case 1:* $x = y = U$: In this case, $x_T = y_T = 1$ and $x_F = y_F = 0$, and thus $z_T = 1, z_F = 0$ and $z = \mathsf{Tr}_{2\to3}(z_T, z_F) = \mathsf{Tr}_{2\to3}(1,0) = U$, as required.
   (b) *Case 2:* $x = T$ and $y = U$: In this case, $x_T = x_F = y_T = 1$ and $y_F = 0$, and thus $z_T = 1$ and $z_F = 0$, implying that $z = U$, as required.
   (c) *Case 3:* $x = U$ and $y = T$: This case is symmetric to the previous case and so also results in U, as required.

It remains to prove that $\mathsf{Tr}_F(\neg_3)$ is correct:

1. If $x = T$, then $x_T = x_F = 1$ and so $z_T = z_F = 0$. Thus, $z = \mathsf{Tr}_{2\to3}(0,0) = F$, as required.
2. If $x = F$, then $x_T = x_F = 0$ and so $z_T = z_F = 1$. Thus, $z = \mathsf{Tr}_{2\to3}(1,1) = T$, as required.
3. If $x = U$, then $x_T = 1$ and $x_F = 0$ and so $z_T = \neg x_F = 1$ and $z_F = \neg x_T = 0$. Thus, $z = \mathsf{Tr}_{2\to3}(1,0) = U$, as required.

Efficiency. The transformations above are very efficient and require 2 Boolean AND gates for every 3VL-AND (or 3VL-OR) gate, and 2 Boolean NOT gates for each 3VL-NOT gate. Using the garbling scheme of [12], this means 4 ciphertext for each $\wedge_3, \vee_3$ gate, and 0 ciphertexts for $\neg_3$ gates. This is far more efficient than any of the previous encodings. However, as we have mentioned above, we still need to show how to compute $\oplus_3$ gates.

4.2 An Efficient Function Transformation for $\oplus_3$ Gates

We now present the transformation for $\oplus_3$ gates for the above functional relation. We begin by remarking that the method above for $\wedge_3$ gates does not work for $\oplus_3$ gates.

x	y	$x \oplus_3 y$	x_T	x_F	y_T	y_F	(z_T, z_F)	z
F	F	F	0	0	0	0	$(0,0)$	F
F	U	U	0	0	1	0	$(1,0)$	U
F	T	T	0	0	1	1	$(1,1)$	T
U	F	U	1	0	0	0	$(1,0)$	U
U	U	U	1	0	1	0	$(0,0)$	F
U	T	U	1	0	1	1	$(0,1)$	`undefined`
T	F	T	1	1	0	0	$(1,1)$	T
T	U	U	1	1	1	0	$(0,1)$	`undefined`
T	T	F	1	1	1	1	$(0,0)$	F

Fig. 3. The result of the transformation of $\oplus_3$ by $(z_T, z_F) = (x_T \oplus y_T, x_F \oplus y_F)$

For example, if we define $z_T = x_T \oplus y_T$ and $z_F = x_F \oplus y_F$, then the result is correct as long as neither of x or y are unknown: If both are unknown then $x = y$, and thus $z_T = z_F = 0$. The result of transforming $(z_T, z_F) = (0,0)$ back to a 3VL is F rather than U. If only one is unknown then $x \neq y$, and thus $z_T = 0$ and $z_F = 1$). The result of transforming $(z_T, z_F) = (0,1)$ is undefined since the pair $(0,1)$ is not in the range of $R_{3\to 2}$. In general, the truth table for the transformation $z_T = x_T \oplus y_T$ and $z_F = x_F \oplus y_F$ appears in Fig. 3; the blue lines are where this transformation is incorrect.

Our transformation must therefore "fix" the incorrect rows in Fig. 3. We define $\mathsf{Tr}_F(\oplus_3)$ that outputs the function $\oplus_2(x_T, x_F, y_T, y_F) = (z_T, z_F)$ defined by

$$\begin{aligned} z'_T &= (x_T \oplus y_T) \oplus \big((x_T \oplus x_F) \wedge (y_T \oplus y_F)\big) \quad \text{and} \quad z'_F = x_F \oplus y_F \\ \mathsf{aux} &= \neg z'_T \wedge z'_F \\ z_T &= z'_T \oplus \mathsf{aux} \quad \text{and} \quad z_F = z'_F \oplus \mathsf{aux} \end{aligned}$$

Observe that the value (z'_T, z'_F) is just the transformation in Fig. 3, with the addition that z'_T is adjusted so that it is flipped in the case that both $x = y = U$ (since in that case $x_T \neq x_F$ and $y_T \neq y_F$). This therefore fixes the 5th row in Fig. 3 (i.e., the input case of $x = y = U$). Note that it doesn't affect any other input cases since $(x_T \oplus x_F) \wedge (y_T \oplus y_F)$ equals 0 in all other cases.

In order to fix the 6th and 8th rows in Fig. 3, it is necessary to adjust the output in the case that $(0,1)$ is received, and only in this case (note that this is only received in rows 6 and 8). Note that the aux variable is assigned value 1 if and only if $z'_T = 0$ and $z'_F = 1$. Thus, defining $z_T = z'_T \oplus \mathsf{aux}$ and $z_F = z'_F \oplus \mathsf{aux}$ adjusts $(z'_T, z'_F) = (0,1)$ to $(z_T, z_F) = (1,0)$ which represents U as required. Furthermore, no other input cases are modified and so the resulting function is correct.

Correctness. The formal proof that this is a valid encoding is demonstrated simply via the truth tables of each encoding. This can be found in Appendix C.2.

Efficiency. The transformation of $\oplus_3$ incurs a cost of two Boolean $\wedge$ gates and 6 Boolean $\oplus$ gates. Utilizing free-XOR and the garbling scheme of [12], we have that 4 ciphertexts are required for garbling $\oplus_3$ gates.

Combining this with Sect. 4.1, we have a cost of 4 ciphertexts for $\wedge_3$ and $\oplus_3$ gates, and 0 ciphertexts for $\neg_3$ gates. This is far more efficient than the naive garbling of Sect. 1.3 for all gate types. Next, recall that the natural encoding of Sect. 3 required 12 ciphertexts for $\wedge_3$ gates and 2 ciphertexts for $\oplus_3$ gates. Thus, denoting by $C_\wedge$ and $C_\oplus$ the number of $\wedge_3$ and $\oplus_3$ gates, respectively, in a 3VL circuit C, we have that the scheme in this section is more efficient if and only if $4 \cdot C_\wedge + 4 \cdot C_\oplus < 12 \cdot C_\wedge + 2 \cdot C_\oplus$, which holds if and only if $C_\oplus < 4 \cdot C_\wedge$. Thus, the natural encoding is only better if the number of $\oplus_3$ gates is *over four times* the number of $\wedge_3$ gates in the circuit. In Sect. 5, we present transformations that perform better in some of these cases.

5 Encoding Using a Non-functional Relation

In this section, we present an alternative encoding that is more expensive for $\wedge_3$ gates but cheaper for $\oplus_3$ gates, in comparison to the encoding of Sect. 4. The value encoding that we use in this section is the same as in Sect. 4, except that we also include $(0,1)$ in the range; thus the relation is no longer functional. Since the motivation regarding the relation is the same as in Sect. 4, we proceed directly to define the relation:

$$R_{3\to 2} = \big\{\big(T,(1,1)\big),\ \big(F,(0,0)\big),\ \big(U,(0,1)\big),\ \big(U,(1,0)\big)\big\}.$$

Thus, $R_{3\to 2}$ maps the 3VL value U to both Boolean pairs $(0,1)$ and $(1,0)$. As such, there are two admissible input transformation functions $\mathsf{Tr}_{3\to 2}$. Both of them map T to $(1,1)$ and map $(0,0)$ to F; one of them maps U to $(1,0)$ the other maps U to $(0,1)$. Recall that our function transformation needs to work for both, in order for the composition theorem to hold.

We use the same notation of (x_T, x_F) as in Sect. 4 for the Boolean pairs in the range of $R_{3\to 2}$. The motivation is the same as before; if $x = T$ or $x = F$ then both values are the same; if $x = U$ then the "true" bit x_T is different from the "false" bit x_F.

The transformation Tr_F for each gate type is given below.

1. $\mathsf{Tr}_F(\wedge_3)$ outputs the function $\wedge_2(x_T, x_F, y_T, y_F) = (z_T, z_F)$, defined by:

$$\begin{aligned} z_T &= x_T \wedge y_T \\ z_F &= (x_F \wedge y_F) \oplus \Big((x_T \oplus x_F) \wedge (y_T \oplus y_F) \wedge \big(\neg(x_F \oplus y_T)\big) \Big) \end{aligned}$$

   Recall that in Sect. 4, it sufficed to define $z_T = x_T \wedge y_T$ and $z_F = x_F \wedge y_F$. However, this does not yield a correct result in this encoding in the case that x and y are both unknown, and x is encoded as $(0,1)$ and y is encoded as $(1,0)$. Specifically, in this case, z is computed as F instead of as U. We fix this case by changing the second bit of z (i.e., z_F) when the encodings are of this form. Observe that the expression $(x_T \oplus x_F) \wedge (y_T \oplus y_F) \wedge (\neg(x_F \oplus y_T))$ evaluates to 1 if and only if $x_T \neq x_F$ and $y_T \neq y_F$ and $x_F = y_T$, which is exactly the case that one of the value is encoded as $(1,0)$ and the other is encoded as $(0,1)$.

2. $\mathsf{Tr}_F(\oplus_3)$ outputs the function $\oplus_2(x_T, x_F, y_T, y_F) = (z_T, z_F)$, defined by:

$$\begin{aligned} z_T &= (x_T \oplus y_T) \oplus \big((x_T \oplus x_F) \wedge (y_T \oplus y_F)\big) \\ z_F &= x_F \oplus y_F \end{aligned}$$

This is the same transformation of $\oplus_3$ described in Sect. 4.2 for the functional encoding of Sect. 4, except that here there is no need to switch the left and right bits of the result in the case that they are $(0, 1)$. This is due to the fact that $(0, 1)$ is a valid encoding of U under $R_{3\to 2}$ used here.

3. $\mathsf{Tr}_F(\neg_3)$ outputs the function $\vee_2(x_T, x_F) = (z_T, z_F)$, defined by:

$$z_T = \neg x_T \quad \text{and} \quad z_F = \neg x_F$$

This is almost the same as the transformation of $\neg_3$ in Sect. 4.1, excepts that we do not exchange the order of the bits. Again, this is due to the fact that both $(1, 0)$ and $(0, 1)$ are valid encodings of 0 and so the negation of U by just complementing both bits results in U and is correct.

Correctness. The formal proof that this is a valid encoding is demonstrated simply via the truth tables of each encoding. This can be found in Appendix C.3.

Efficiency. The Boolean function $\mathsf{Tr}_F(\wedge_3)$ requires 4 AND gates, which translates to 8 ciphertexts using the garbling of [12]. The Boolean function $\mathsf{Tr}_F(\oplus_3)$ requires only one AND gate, which translates to two ciphertexts using the garbling of [12]. Denote by $C_\wedge$ and $C_\oplus$ the number of $\wedge_3$ and $\oplus_3$ gates in the 3VL circuit, then the encoding of this section is better than that of Sect. 4 if and only if $8 \cdot C_\wedge + 2 \cdot C_\oplus < 4 \cdot C_\wedge + 4 \cdot C_\oplus$ which holds if and only if $C_\oplus > 2 \cdot C_\wedge$. Observe also that the encoding in this section is always at least as good as the natural encoding of Sect. 3; in particular, it has the same cost for $\oplus_3$ gates and is strictly cheaper for $\wedge_3$ gates.

6 Efficiency Summary of the Different Methods

We have presented a naive garbling method and three different encodings. We summarize the efficiency of these different methods, as a function of the number of ciphertexts needed when garbling, in Table 2.

Table 2. A summary of the garbling efficiency of the different methods

Encoding	Ciphertexts for $\wedge_3$	Ciphertexts for $\oplus_3$	Best in range
Section 1.3 – Naive	8	8	None
Section 3 – Natural	12	2	None
Section 4 – Functional	4	4	$C_\oplus < 2 \cdot C_\wedge$
Section 5 – Non-functional	8	2	$C_\oplus > 2 \cdot C_\wedge$

7 A Black-Box Protocol for Computing 3VL Circuits

In this section, we show how to securely compute 3VL circuits. Of course, one could design a protocol from scratch using a garbled 3VL circuit. However, our goal is to be able to use *any* protocol that can be used to securely evaluate a Boolean circuit, and to directly inherit its security properties. This approach is simpler, and allows us to leverage existing protocol optimizations for the Boolean case.

Before proceeding, we explain why there is an issue here. Seemingly, one could compile any 3VL-circuit into a Boolean circuit using our method above, and then run the secure computation protocol on the Boolean circuit to obtain the output. As we will see, this is actually not secure. Fortunately, however, it is very easy to fix. We now explain why this is not secure:

1. *Output leakage:* The first problem that arises is due to the fact that Definition 2.1 allows $R_{3\to 2}$ to be a non-functional relation. This implies that a value $x \in \{T, F, U\}$ might be mapped to two or more Boolean representations. Now, if a secure protocol is run on the Boolean circuit, this implies that a single 3VL output could be represented in more than one way. This could potentially leak information that is not revealed by the function itself. In Appendix B, we show a concrete scenario where this does actually reveal more information than allowed. We stress that this leakage can occur even if the parties are *semi-honest*.
   This leakage can be avoided by simply transforming y to a *unique*, predetermined Boolean value y^* at the end of the circuit computation and before outputs are revealed. This is done by incorporating an "output translation" gadget into the circuit for every output wire.
2. *Insecurity due to malicious inputs.* Recall that the relation $R_{3\to 2}$ does not have to be defined over the entire range of $\{0, 1\} \times \{0, 1\}$, and this is indeed the case for the relation that we use in Sect. 4. In such a case, if the malicious party inputs a Boolean input that is not legal (i.e., is not in the range of $R_{3\to 2}$), then this can result in an incorrect result (or worse).
   This cheating can be prevented by incorporating an "input translation" gadget for every input wire of the circuit that deterministically translates all possible Boolean inputs (even invalid ones) into valid inputs that appear in the range of $R_{3\to 2}$. This prevents a malicious adversary from inputting incorrect values (to be more exact, it can input incorrect values but they will anyway be translated into valid ones).

The key observation from above is that the solutions to both problems involve modifications to the circuit only. Thus, *any* protocols that is secure for arbitrary Boolean circuits can be used to securely compute 3VL circuits. Furthermore, these input and output gadgets are very small (at least for all of our encodings) and thus do not add any significant overhead.

We have the following theorem[6]:

Theorem 7.1. *Let π be a protocol for securely computing any Boolean circuit, let f_3 be a 3VL function with an associated 3VL circuit C, and let C' be a Boolean circuit that is derived from C via a valid 3VL-Boolean encoding. Then, Denote by C'_1 the circuit obtained by adding* output-translation *gadgets to C', and denote by C'_2 the circuit obtained by adding* input-translation *and* output-translation *gadget to C'.*

1. *If π is secure in the presence of semi-honest adversaries, then protocol π with circuit C'_1 securely computes the 3VL function f_3 in the presence of semi-honest adversaries.*
2. *If π is secure in the presence of malicious (resp., covert) adversaries, then protocol π with circuit C'_2 securely computes the 3VL function f_3 in the presence of malicious (resp., covert) adversaries.*

Secure computation. The above theorem holds for *any* protocol for secure computation. This includes protocols based on Yao and garbled circuits [8,13], as well as other protocols like that of [4].

8 Lower Bounds

One of the most important optimizations of the past decade for garbled circuits is that of free-XOR [7]. Observe that none of the 3VL-Boolean encodings that we have presented have free-XOR, and the cheapest transformation of $\oplus_3$ requires 2 ciphertexts. In this section, we ask the following question:

Can free-XOR garbling be achieved for 3VL functions?

We prove a negative answer for a *linear garbling scheme*, which is defined in the Linicrypt model of [1]. Our proof is based on a reduction from any garbling scheme for 3VL circuit to a garbling scheme for Boolean circuits. Specifically, we show that any garbling scheme for 3VL-XOR can be used to garble Boolean-AND gates at the exact same cost. Now, [12] proved that at least 2 ciphertexts are required for garbling AND gates using any linear garbling method. By reducing to this result, we will show that 3VL-XOR cannot be garbled with less than two ciphertexts using any linear garbling method. Thus, a significant breakthrough in garbling would be required to achieve free-XOR in the 3VL setting, or even to reduce the cost of 3VL-XOR to below two ciphertexts.

Reducing Boolean AND to 3VL XOR. It is actually very easy to compute a Boolean AND gate given a 3VL XOR gate. This is due to the fact that 3VL XOR actually contains an *embedded AND*; this is demonstrated in Fig. 4.

[6] The proof of the theorem is straightforward and is thus omitted.

This can be utilized in the following way. Let $\tilde{g}$ be a garbled 3VL-XOR gate with input wires x, y and output wire z. By definition, given keys k_x^α and k_y^β on the input wires with $\alpha, \beta \in \{T, F, U\}$, the garbled gate can be used to compute the key k_z^γ on the output wire where $\gamma = \alpha \oplus_3 \beta$. Thus, in order to compute a Boolean AND gate, the following can be carried out. First, associate the 3VL-value F with the Boolean value 1 (True), and associate the 3VL-value U with the Boolean value 0 (False). Then, given any two of k_x^U, k_x^F and k_y^U, k_y^F the output of the garbled gate will be k_z^F if and only if $x = y = F$, which is exactly a Boolean AND gate. (This is depicted in the shaded square in Fig. 4.) Observe that the 3VL-value T is not used in this computation and so is ignored. The fact that this method is a *secure* garbling of an AND gate follows directly from the security of the 3VL garbling scheme.

$\oplus_3$	**T**	**F**	**U**
T	F	T	U
F	T	F	U
U	U	U	U

Fig. 4. Shows that $\oplus_3$ embeds the truth table of both $\wedge, \vee$ and $\oplus$.

It follows that a (single) Boolean AND gate can be garbled at the same cost of a 3VL XOR gate. Thus, free-3VL-XOR would imply free-Boolean-AND, and even 3VL XOR with just a single ciphertext would imply a construction for garbling a Boolean AND gate at the cost of just one ciphertext. Both of these would be surprising results. We now formalize this more rigorously using the framework of linear garbling.

Impossibility for linear garbling. The notion of linear garbling was introduced by [12], who also showed that all known garbling schemes are linear. In their model, the garbling and evaluation algorithms use only linear operations, apart from queries to a random oracle (which may be instantiated by the garbling scheme) and choosing which linear operation to apply based on some select bits for a given wire. They prove that for every ideally secure linear garbling scheme (as defined in [12]), at least two ciphertexts must be communicated for every Boolean AND gate in the circuit. Combining [12, Theorem 3] with what we have shown above, we obtain the following theorem with regards to garbling schemes for 3VL circuits in the same model.

Theorem 8.1. *Every ideally secure garbling scheme for 3VL-XOR gates, that is linear in the sense defined in [12], has the property that the garbled gate consists of at least* $2n$ *bits, where* n *is the security parameter.*

This explains why we do not achieve free-XOR in our constructions in the three-valued logic setting.

A Exhaustive Search for Expressions with One Boolean AND

We present a simplified version of the technique that we used in order to search for a Boolean expression with only one AND gate (and an unlimited number of XOR gates) that implements the functionality of a 3VL AND and 3VL XOR.

We actually used several simple optimizations to this technique to make it run faster, but these are not of significance to the discussion and so are omitted.

In this paper we focus on a specific set of possible 3VL-to-Boolean encodings, specifically, we focus on encodings that map each 3VL value x (i.e. T, F, U) to a pair of Boolean values (x_L, x_R) (L, R for left and right). Note that this means that either the encoding is *functional*, which means that each 3VL value is mapped to exactly one Boolean pair and hence there remains one invalid Boolean pair, or the encoding is *non-functional* which means that one 3VL value is mapped to two Boolean pairs while the other two 3VL values are mapped to a single Boolean pair. The total number of possible encodings (functional and non-functional) is 60, as can be seen by a simple combinatorical computation.

Let Enc be some 3VL-to-Boolean encoding. A Boolean implementation of 3VL-AND (resp. 3VL-XOR) using Enc is given two pairs of Boolean values, (x_L, x_R) and (y_L, y_R), and outputs a single pair of Boolean values (z_L, z_R), such that when given encodings of the 3VL values x and y it outputs an encoding of $x \wedge_3 y$ (resp. $x \oplus_3 y$) where $\wedge_3$ is a 3VL-AND (resp. $\oplus_3$ is a 3VL-XOR). When Enc is non-functional, this should hold for every possible encoding of x and y. This means that for a functional encoding we test the correctness of 9 possibilities, and for a non-functional encoding we test the correctness of 16 possibilities of (x_L, x_R) and (y_L, y_R).

Since we are interested in an implementation with a single Boolean AND gate, the values of z_L and z_R are basically a Boolean expression over the four literals x_L, x_R, y_L, y_R and the constant 1 (the constant 0 can be obtained by simply XORing the literal with itself) with a single Boolean AND and unlimited number of Boolean XOR gates. Our goal is to find a way to enumerate over all these expressions and test if they form a correct implementation of the 3VL-AND and 3VL-XOR.

The exhaustive search process is depicted in Fig. 5. We set $E_0 = \{x_L, x_R, y_L, y_R, 1\}$. This is the set of initial values, from which the expressions for z_L, z_R are formed. Before we apply a Boolean AND to the above values we first want to obtain a set of all possible expressions using Boolean XOR gates only, we denote this set by E_0^+. Note that E_0^+ can be obtained by taking the XOR of each non-empty subset of values from E_0, which means that $|E_0^+| = |E_0| + \sum_{i=1}^{5} \binom{5}{i} = 36$. Then, we can choose 2 expressions $e_1, e_2 \in E_0^+$ and apply $e_3 = e_1 \wedge e_2$. We denote the set of possible expressions of this form by E_1 and by counting we get that $E_1 = \binom{|E_0^+|}{2} = \binom{36}{2} = 630$. Notice that E_1 does not contain *all* possible expressions with exactly one Boolean AND, since XOR operations are only computed before the AND. Thus, for example, $x_L \oplus (x_R \wedge y_L) \notin E_1$. In order to obtain the

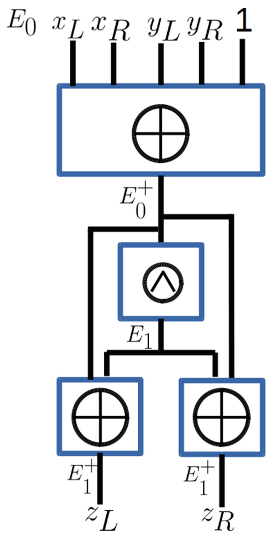

Fig. 5. The exhaustive search process.

set of *all* possible expressions, denoted E_1^+, we need to add another layer of Boolean XORs after the AND. However, since we want to test *pairs* of Boolean expressions for z_L, z_R using up to one Boolean AND gate, we may use the *same* expression from E_1 and apply XOR after it with two different expressions from E_0^+. Therefore, we have that $E_1^+ = E_1 \times (E_0^+ \times E_0^+)$. Note that E_1^+ also contains expressions with no Boolean AND at all. For example, for any expression e without Boolean AND gates, E_1^+ also contains $e = (1 \wedge 1) \oplus (\neg e)$. We therefore conclude that $|E_1^+| = |E_1| \cdot |E_0^+|^2 = 630 \cdot 36^2 = 816480$.

Putting it all together, we have 60 possible encodings. For each encoding, we have 816,480 possible pairs of expressions for z_l, z_R with up to one AND gate and for each possible pair we need to test its correctness over 9 or 16 possible inputs. The total number of tests is therefore $60 \cdot 816480 \cdot 9 = 440,899,200 \approx 2^{28.7}$ or $60 \cdot 816480 \cdot 16 = 783,820,800 \approx 2^{29.5}$.

B Insecurity of the Naive Protocol for Evaluating 3VL Functions

In this section we provide a concrete attack on a protocol that uses a valid Boolean encoding, without adding the input/output gadgets described in Sect. 7. Consider the 3VL function $f_3 : \{T, F, U\}^2 \to \{T, F, U\}^2$ defined by $f_3(a, b) = a \oplus_3 b$; denote the output by c. Now, consider a 2-party protocol for evaluating this function, where P_1 inputs both a and b, and P_2 does not input anything. (Needless to say, this is a silly example since such a function can be singlehandedly computed by P_1 and the result can be sent to P_2. However, this illustrates the problem, and of course applies to more "interesting" cases as well.)

Now, assume that the output of the function is U. In this case, a secure evaluation of f does not reveal anything to P_2 except the fact that (a, b) is either (U, U), (T, U), (U, T), (F, U) or (U, F). Furthermore, consider the case that P_1's inputs are random. In this case, each of these possible inputs occurs with probability $\frac{1}{5}$ (assuming P_2 has no auxiliary information). Consider now what happens if a secure two-party protocol is run to compute this function on the encoding, without applying an output transformation gadget as described in Sect. 7. For the sake of concreteness, consider the non-functional relation encoding of Sect. 5. For this encoding U can be mapped to $(1, 0)$ or to $(0, 1)$. Assume that $\mathsf{Tr}_{3\to 2}(U) = (0, 1)$. Then, the possible outputs of the function (since it is just a single XOR) are given in Table 3; the shaded rows are associated with output U:

Observe that if P_2 receives $(z_T, z_F) = (0, 1)$ for output, then it *knows* that P_1's input was either (F, U) or (U, F). In contrast, if P_2 receives $(z_T, z_F) = (1, 0)$ for output, then it *knows* that P_1's input was either (U, U) or (U, T) or (T, U). This is clearly information that P_2 should not learn (observe also that if P_2 receives $(0, 1)$ then it know with full certainty that either $a = F$ or $b = F$). This is therefore not a secure protocol.

Table 3. $\mathsf{Tr}_F(\oplus_3)$

x	y	z	x_T	x_F	y_T	y_F	(z_T, z_F)	z
F	F	F	0	0	0	0	$(0,0)$	F
F	U	U	0	0	0	1	$(0,1)$	U
F	T	T	0	0	1	1	$(1,1)$	T
U	F	U	0	1	0	0	$(0,1)$	U
U	U	U	0	1	0	1	$(1,0)$	U
U	T	U	0	1	1	1	$(1,0)$	U
T	F	T	1	1	0	0	$(1,1)$	T
T	U	U	1	1	0	1	$(1,0)$	U
T	T	F	1	1	1	1	$(0,0)$	F

C Formal Proofs of Encodings via Truth Tables

In this appendix, we provide the truth tables for each of our encoding methods. These truth tables constitute a formal proof of correctness, since they show that the mapping from input to output is correct for all possible inputs.

C.1 Correctness of the Natural Encoding

See Tables 4, 5 and 6.

Table 4. The Boolean encoding of 3VL-AND in Sect. 3

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	$(0,0)$	F
F	U	F	0	0	0	1	$(0,0)$	F
F	T	F	0	0	1	0	$(0,0)$	F
F	U	F	0	0	1	1	$(0,0)$	F
U	F	F	0	1	0	0	$(0,0)$	F
U	U	U	0	1	0	1	$(0,1)$	U
U	T	U	0	1	1	0	$(0,1)$	U
U	U	U	0	1	1	1	$(0,1)$	U
T	F	F	1	0	0	0	$(0,0)$	F
T	U	U	1	0	0	1	$(0,1)$	U
T	T	T	1	0	1	0	$(1,0)$	T
T	U	U	1	0	1	1	$(1,1)$	U
U	F	F	1	1	0	0	$(0,0)$	F
U	U	U	1	1	0	1	$(0,1)$	U
U	T	U	1	1	1	0	$(1,1)$	U
U	U	U	1	1	1	1	$(1,1)$	U

Table 5. The Boolean encoding of 3VL-XOR in Sect. 3

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	$(0,0)$	F
F	U	U	0	0	0	1	$(0,1)$	U
F	T	T	0	0	1	0	$(1,0)$	T
F	U	U	0	0	1	1	$(1,1)$	U
U	F	U	0	1	0	0	$(0,1)$	U
U	U	U	0	1	0	1	$(0,1)$	U
U	T	U	0	1	1	0	$(1,1)$	U
U	U	U	0	1	1	1	$(1,1)$	U
T	F	T	1	0	0	0	$(1,0)$	T
T	U	U	1	0	0	1	$(1,1)$	U
T	T	F	1	0	1	0	$(0,0)$	F
T	U	U	1	0	1	1	$(0,1)$	U
U	F	U	1	1	0	0	$(1,1)$	U
U	U	U	1	1	0	1	$(1,1)$	U
U	T	U	1	1	1	0	$(0,1)$	U
U	U	U	1	1	1	1	$(0,1)$	U

Table 6. The Boolean encoding of 3VL-NOT in Sect. 3

x	$\neg_3(x)$	x_T	x_U	(z_T, z_U)	z
F	T	0	0	$(1,0)$	T
U	U	0	1	$(1,1)$	U
T	F	1	0	$(0,0)$	F
U	U	1	1	$(0,1)$	U

C.2 Correctness of the Encoding Using a Functional Relation

See Tables 7, 8 and 9.

Table 7. The Boolean encoding of 3VL-AND in Sect. 4

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	(0, 0)	F
F	U	F	0	0	1	0	(0, 0)	F
F	T	F	0	0	1	1	(0, 0)	F
U	F	F	1	0	0	0	(0, 0)	F
U	U	U	1	0	1	0	(1, 0)	U
U	T	U	1	0	1	1	(1, 0)	U
T	F	F	1	1	0	0	(0, 0)	F
T	U	U	1	1	1	0	(1, 0)	U
T	T	T	1	1	1	1	(1, 1)	T

Table 8. The Boolean encoding of 3VL-XOR in Sect. 4

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	(0, 0)	F
F	U	U	0	0	1	0	(1, 0)	U
F	T	T	0	0	1	1	(1, 1)	T
U	F	U	1	0	0	0	(1, 0)	U
U	U	U	1	0	1	0	(1, 0)	U
U	T	U	1	0	1	1	(1, 0)	U
T	F	T	1	1	0	0	(1, 1)	T
T	U	U	1	1	1	0	(1, 0)	U
T	T	F	1	1	1	1	(0, 0)	F

Table 9. The Boolean encoding of 3VL-NOT in Sect. 4

x	$\neg_3(x)$	x_T	x_U	(z_T, z_U)	z
F	T	0	0	(1, 1)	T
U	U	1	0	(1, 0)	U
T	F	1	1	(0, 0)	F

C.3 Correctness of the Encoding Using a Non-functional Relation

See Tables 10, 11 and 12.

Table 10. The Boolean encoding of 3VL-AND in Sect. 5

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	(0, 0)	F
F	U	F	0	0	0	1	(0, 0)	F
F	U	F	0	0	1	0	(0, 0)	F
F	T	F	0	0	1	1	(0, 0)	F
U	F	F	0	1	0	0	(0, 0)	F
U	U	U	0	1	0	1	(0, 1)	U
U	U	U	0	1	1	0	(0, 1)	U
U	T	U	0	1	1	1	(0, 1)	U
U	F	F	1	0	0	0	(0, 0)	F
U	U	U	1	0	0	1	(0, 1)	U
U	U	U	1	0	1	0	(1, 0)	U
U	T	U	1	0	1	1	(1, 0)	U
T	F	F	1	1	0	0	(0, 0)	F
T	U	U	1	1	0	1	(0, 1)	U
T	U	U	1	1	1	0	(1, 0)	U
T	T	T	1	1	1	1	(1, 1)	T

Table 11. The Boolean encoding of 3VL-XOR in Sect. 5

x	y	z	x_T	x_U	y_T	y_U	(z_T, z_U)	z
F	F	F	0	0	0	0	(0, 0)	F
F	U	U	0	0	0	1	(0, 1)	U
F	U	U	0	0	1	0	(1, 0)	U
F	T	T	0	0	1	1	(1, 1)	T
U	F	U	0	1	0	0	(0, 1)	U
U	U	U	0	1	0	1	(1, 0)	U
U	U	U	0	1	1	0	(0, 1)	U
U	T	U	0	1	1	1	(1, 0)	U
U	F	U	1	0	0	0	(1, 0)	U
U	U	U	1	0	0	1	(0, 1)	U
U	U	U	1	0	1	0	(1, 0)	U
U	T	U	1	0	1	1	(0, 1)	U
T	F	T	1	1	0	0	(1, 1)	T
T	U	U	1	1	0	1	(1, 0)	U
T	U	U	1	1	1	0	(0, 1)	U
T	T	F	1	1	1	1	(0, 0)	F

Table 12. The Boolean encoding of 3VL-NOT in Sect. 5

x	$\neg_3(x)$	x_T	x_U	(z_T, z_U)	z
F	T	0	0	(1, 1)	T
U	U	0	1	(0, 1)	U
U	U	1	0	(1, 0)	U
T	F	1	1	(0, 0)	F

References

1. Carmer, B., Rosulek, M.: Linicrypt: a model for practical cryptography. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 416–445. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_15
2. de Haan, L., Gennick, J.: Nulls: Nothing to Worry About. Oracle Magazine (2005). http://www.oracle.com/technetwork/issue-archive/2005/05-jul/o45sql-097727.html/

3. Goldreich, O.: Foundations of Cryptography: Volume 2 - Basic Applications. Cambridge University Press, Cambridge (2004)
4. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game - a completeness theorem for protocols with honest majority. In: 19th STOC, pp. 218–229 (1987). For details see [3]
5. ISO/IEC. ISO/IEC 9075-2:2016 (2016). http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=63556/
6. Kleene, S.C.: Introduction to Metamathematics. Bibliotheca Mathematica (1952)
7. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
8. Lindell, Y., Pinkas, B.: A proof of Yao's protocol for secure two-party computation. J. Cryptol. **22**(2), 161–188 (2009)
9. Microsoft. Null and Unknown (Transact-SQL). https://msdn.microsoft.com/en-us/library/mt204037.aspx/
10. MySQL. MySQL 5.7 Reference Manual 13.3.3: Logical Operators. http://dev.mysql.com/doc/refman/5.7/en/logical-operators.html/
11. 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
12. 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. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8
13. Yao, A.: How to generate and exchange secrets. In: 27th FOCS, pp. 162–167 (1986)