# Secure and Trusted Partial White-box Verification Based on Garbled Circuits

# SECURE AND TRUSTED PARTIAL WHITE-BOX VERIFICATION BASED ON GARBLED CIRCUITS

BY

HONGSHENG ZHONG, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

| | |
|---|---|
| Master of Science (2016) | McMaster University |
| (Computig & Software) | Hamilton, Ontario, Canada |

| | |
|---|---|
| TITLE: | Secure and Trusted Partial White-box Verification Based on Garbled Circuits |
| AUTHOR: | Hongsheng Zhong<br>B.Eng., (Information Security Engineering)<br>Harbin Institute of Technology, Harbin, China |
| SUPERVISOR: | Dr. Karakostas, Dr. Wassyng |
| NUMBER OF PAGES: | ix, 101 |

# Abstract

Verification is a process that checks whether a program $G$, implemented by a developer, correctly complies with the corresponding requirement specifications. A verifier, whose interests may be different from the developer, will conduct such verification on $G$. However, as the developer and the verifier distrust each other probably, either of them may exhibit harmful behavior and take advantage of the verification. Generally, the developer hopes to protect the content privacy of the program, while the verifier wants to conduct effective verification to detect the possible errors. Therefore, a question inevitably arises: How to conduct an effective and efficient kind of verification, without breaking the security requirements of the two parties?

We treat verification as a process akin to testing, i.e. verifying the design with test cases and checking the results. In order to make the verification more effective, we get rid of the limitations in traditional testing approaches, like black-box and white-box testing, and propose the "partial white-box verification".

Taking circuits as the description means, we regard the program as a circuit graph. Making the structure of the graph public, we manage to make the verification process in such a graph partially white-box. Via garbled circuits, commitment schemes and other techniques, the security requirements in such verification are guaranteed.

# Acknowledgements

This thesis could not have been completed without the great support from numerous people over the two years.

I would like to show my sincere gratitude to my supervisors, Dr.Karakostas and Dr.Wassyng. I am grateful to them for letting me take part in the research. The support, either academically or financially, provided me such a great opportunity to realize something I could have never been able to do. Their guidance in the working style, the learning habits and the academic attitude, made deep impressions on me, and will be beneficial to my whole life.

Thanks are also due to my previous colleague, Yixian Cai. He built the theoretical basis for my work, and provided some original ideas. In the meantime, he assisted me in researching and understanding about the topics. His help mattered at the beginning stage. Besides, I should acknowledge the contribution of another collaborator, Zhe Ji. He implemented my theoretical work into the Java program.

In addition, I am appreciative of my friends, who offered the help of proofreading and suggestions in both language and mathematics. Particularly, I am also very grateful for the spiritual encouragement from Yiyi, Chen.

Everything would not happen without the excellent platforms. Special mention goes to the Dept. of Computing and Software. Big thanks to the outstanding faculties, the kind staff. I need to express my deepest appreciation to McMaster University. I have an unforgettable time here. These aspects contribute to the accomplishment of my program.

I am indebted to my family and my relatives for supporting me to go abroad. I have owed them a lot during the two years, but they always understood my situation and still helped me.

To anyone that I may have forgotten. I apologize. Thank you as well.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Subcontracting the development of product to a third-party is a common phenomenon nowadays. For example, a finished car consists of various components, which are often designed by different manufacturers. For a automobile manufacturer, he possesses a *specification* of required behavior about the components (e.g, engines, tyres) and subcontracts other manufacturers to design or implement the corresponding units. When the automobile manufacturer obtains the delivery from the subcontractors, an intuitive question is raised: if the finished car assembled by the components, which are implemented by the subcontractors, complies with the specification? Thus, the automobile manufacturer has to *verify* if the functionality of the components matches the specification. To realize this type of system verification is the core of our work.

The verification in the above case involves two problems. The *first problem* is that the subcontractors want to conceal the design and implementation from clients or third-party. The requirement stems from the fact that the subcontractors want to protect the proprietary information of the components. The *second problem* is the limitations in traditional strategies of verifying. The common means of verifying is to test a product with test cases and check if the results match the specification. There are two most typical strategies to conduct a test: “black box” and “white box”. The black-box strategy hides the implementation and only allows the *verifier* observing the external inputs and outputs. In the white-box strategy, the *developer* (namely the subcontractors) can expose everything of implementation. However, both of the strategies cannot work on the example that the automobile manufacturer verifying the subcontracted components. If adopted white-box test strategy, the proprietary information of the components must be revealed; if adopted black-box test strategy, the little information from the external behavior cannot help the verifier find the failed components. From the perspective of the verifier, he always hopes that the verification can be more powerful, even taking advantage of the developer’s information. This fact leads to a dilemma between the information privacy and the power of verification.

In this work, we try to find a balanced way to make the verification adequate and secure. Combining the benefits of the black-box test and white-box test exactly realizes this point. We allow to leak the positions of the components, to enable the verification to perform in a path or a network. So the internal relations can be revealed and tested. But in the meantime, the secure requirements should not be compromised. Therefore, what we try to realize is a secure and trusted “partial white-box verification”.

We are interested in applying such kind of verification to the development of software programs. A software system also consists of many components and these components may be designed/implemented by the subcontractors. A verifier who owns the specification wants to verify if the components in the software systems work well, like what the above automobile manufacturer does. The verification on software systems should not break the privacy of the program too, since the developer (the subcontractors) regards the content of the program as proprietary information. Verifying software systems is often conducted remotely and the two parties, a verifier and the developer, distrust each other. The protocol should satisfy the following properties:

- Correctness - With test cases, the protocol enables both parties to execute the verification with correct procedure and obtain final result complying with the specification. If the execution deviates from the standard procedure, the result should not be correct.

- Security - The protocol should protect the secrets of the developer from leaking to the verifier, specifically, the content of a program. Besides, the protocol should make sure the two parties to obey the correct procedure and avoid malicious behavior.

In this thesis, we will propose a protocol to realize the secure and trusted "partial white-box verification" and satisfy the above properties.

## 1.2 Previous work

Our goal relates to the privacy of information (hiding the content of the program) and trusted computation (honestly following the predefined procedure). Much previous research has been invested in these fields. *Obfuscation* is a method of transforming a program into a form that has the same functionality as the original program but is difficult to reverse engineer. The ideal obfuscation makes the obfuscated program a "virtual black box"–nothing leaked beyond the input-output behavior (Goldwasser and Rothblum, 2007). However, the ideal "black-box" obfuscation is hard to achieve. As a fact was proved in (Barak *et al.*, 2001), a certain family of functionality can always breach the "black-box" property. Although choosing certain classes of functions or relaxing the security properties produces positive results (Garg *et al.*, 2013; Boyle *et al.*, 2014; Wee, 2005), this concept still remains in a theoretical level.

In 2009, Craig Gentry proposed the first *Fully homomorphic encryption* ($FHE$) scheme (Gentry *et al.*, 2009), which makes this powerful technique closer to the practical applications. $FHE$ is a cryptography scheme that supports arbitrary computation on encrypted texts. Based on $FHE$'s powerful features, Yixian Cai made the first attempt to build the secure and trusted "partial white-box verification" (Cai *et al.*, 2016). His work made the following contributions: 1. Cai posed the problems for the first time, and provided the strict definitions of correctness and security; 2. He proposed a protocol to solve the partial white-box verification in the honest and malicious cases. The limitation of $FHE$ is the efficiency. As discussed in (Wang *et al.*, 2015), $FHE$ is restricted by its expensive computational overhead. Although the implementation of $FHE$ has been presented (Gentry and Halevi, 2011), $FHE$ is still not the perfect tool.

An alternative technique to $FHE$ is garbled circuits. *Garbled circuits* (or *Yao garbled circuits* for short) was proposed by Andrew Yao (Yao, 1982, 1986). The security goal of garbled circuits is to protect the privacy of the target circuit and the inputs owned by each party who participates the protocol of garbled circuits (Section 2.4). Recent work provides garbled circuits a mature basis in the aspects of theory and implementation. The first rigid proof about the security of garbled circuits was proposed in (Lindell and Pinkas, 2009). A system, Fairplay (Malkhi *et al.*, 2004), was the first attempt to implement garbled circuits, and it was updated to support multi-parties afterwards (Ben-David *et al.*, 2008). Other important attempts in implementing garbled circuits were made in (Huang *et al.*, 2011; Kreuter *et al.*, 2012).

Nevertheless, the scheme of Yao garbled circuits only works against *semi-honest adversaries*, who honestly follow all required steps but record all the received information (Goldreich, 2004, Chapter 7). In terms of our problem, because both the developer and the verifier may conduct *malicious behavior*, the scheme of garbled circuits must be secure more than against semi-honest adversaries. Much efforts were invested in this field to improve the security of garbled circuits. The work in (Naor and Pinkas, 2001, 2005) proposed an advanced *oblivious transfer* protocol (or *OT*, Section 2.5) that is secure in the malicious case. A solution against malicious behavior based on zero-knowledge proof was designed in (Goldreich *et al.*, 1987), but it is not practical concerning the computational inefficiency. A more practical and well-known way is a *cut-and-choose* strategy. The idea behind this approach is that each circuit should be garbled multiple times. Consequently, each circuit corresponds to more than one garbled circuits. Among these circuits, some of them can be used to check the data integrity. This method eliminates the success chance when the protocol fails in detecting corrupt circuits. The cut-and-choose approach has different variants to defend malicious behavior. The work in (Shen *et al.*, 2011) proposed a solution for two-output function in malicious case. Mohassel (Mohassel and Riva, 2013) designed a protocol with malicious security by two stages of cut-and-choose checking.

Particularly, Lindell and Pinkas (Lindell and Pinkas, 2007) presented the most comprehensive explanation about cut-and-choose strategy against malicious case. Their work provided an efficient paradigm, which is adopted in our construction. Apart from the above approaches, other researchers also provided original ideas to make garbled circuits secure against malicious behavior (Nielsen and Orlandi, 2009).

In terms of checking the data integrity of computation results, *verifiable computation*($VC$) is a concept related to our problems, which responds a question: *how can we trust a result computed by a third party?* (Walfish and Blumberg, 2015). In other words, $VC$ can be used to check the integrity of computation. Intuitively, repeating the computation is the simplest way to realize VC. But the double cost of execution is too expensive, and the credibility of the second execution cannot be guaranteed. Another idea is to return the computational result with a proof. This approach is the *proof-based verifiable computation* which employs *probabilistic checkable proofs* and *interactive proofs* from complexity theory. Although the work in (Ben-Sasson *et al.*, 2013; Braun *et al.*, 2013; Setty *et al.*, 2013; Parno *et al.*, 2016) has proposed implemented systems, the problem of inefficiency and incomplete expressiveness (the class of computations that the system supports) is still not solved. Nonetheless, $VC$ is still a valuable subject related to our work.

## 1.3 Our contributions

We build a cryptographic construction to realize the secure and trusted "partial white-box verification". Our work takes garbled circuits as the fundamental tool. Our contributions are as following:

1. We propose a theoretical implementation satisfying the properties of security and correctness. The program is decomposed into many modules which are represented by a standard means of description, *tabular expressions*. The theoretical implementation contains algorithmic construction (Section 3.6) and definitions of the cryptographic schemes (Section 3.3.2). Through breaking down the requirements of security and correctness, we realize three specific targets:

   - *Protecting the privacy of program:* We present a protocol which protects the content of the program from leaking to other parties except the developer.

     The protocol of garbled circuits guarantees the content privacy towards the target circuits, and thus we apply this tool to encrypt the program. Particularly, the original program should be transformed from tabular expressions into *circuits* (Section 2.1). The security of garbled circuits can be reduced to the security of pseudorandom generators (Section 2.3).

   - *Protecting the real values of intermediate results:* An *intermediate result* indicate the output of computing a certain circuit and becomes the input for other circuits. In partial white-box verification, the leakage of intermediate results may expose the secrets. When a program is transformed into many circuits and these circuits are encrypted into garbled circuits, each circuit has different *garbled keys* to encode its inputs and outputs. If a value is communicated between two garbled circuits, it would need a step of "translation". For instance, when a circuit $CT_u$ generates a computational result $y$, $y$ is encoded by $CT_u$'s garbled keys. In this case, another circuit $CT_v$ cannot directly compute $y$, because $y$ is encoded by different garbled keys. The step of "translation" means that the value encrypted by the garbled keys should be decrypted first, and then encrypted by another set of keys.

The real values behind the encrypted intermediate results often contains some sensitive information, so the step of "translation" should also protect the real values from exposing. However, this requirement is a challenge to the protocol garbled circuits. Inspired by the related work (Bellare *et al.*, 2012; Snyder, 2014; Mohassel and Riva, 2013), we design an algorithm $TF$ (Section 3.5.1) to solve this problem.

- *Being secure in the malicious case:* In our problem, both the developer and the verifier can conduct arbitrary behavior to break the security requirements. To avoid this type of threats, the protocol should be secure in the malicious case. As introduced in Section 1.2, we employ the cut-and-choose approach to enhance the security of garbled circuits and make the protocol secure against malicious behavior. Our work incorporates the paradigm in (Lindell and Pinkas, 2007). According to the requirements of our problem, we make some changes and design different mechanisms to check the data integrity, while the security is not compromised. The description about the cut-and-choose approach is in Section 2.6, and the details about how to incorporate it into our construction is located in Section 3.6.

  Admittedly, the construction based on $FHE$ in (Cai *et al.*, 2016) provides stronger capability to detect malicious behavior than our protocol, as the former allows the protocol to conduct a reverse computation to detect the wrong results. Nevertheless, our protocol provides a more efficient and reliable solution. The comparison between our protocol with the $FHE$'s construction is in Section 3.2.

2. Feasibility is an important aspect in our work. Derived the framework from (Lindell and Pinkas, 2007), our construction also has the particular advantages in implementation.

   In Chapter 3, we specify the design of our framework and present algorithms in pseudocode. These algorithms, we believe, are sufficiently detailed and the pseudocode can be directly implemented without consulting secondary references. The illustration in Section 3.6.2 can be easily transformed into an interaction diagram for a design specification. Actually, based on the framework of Fairplay (Malkhi *et al.*, 2004), my colleague Ji has implemented our construction in Java (Ji, 2016). This inspiring news gains our confidence in assessing our work.

## 1.4 Organization

In Chapter 2, we introduce the preliminaries which are used in the following construction, e.g. the concept of tabular expressions, garbled circuits and the cut-and-choose approach. In Chapter 3, we present the definitions and construction for the secure and trusted "partial white-box verification" protocol. In Section 3.7 and 3.8, the proof of correctness and security for our construction is presented. In Chapter 4, open questions are provided for future study.

# Chapter 2

# Preliminaries

## 2.1 Means of Describing the Program

### 2.1.1 Tabular Expressions

A *tabular expression*, or called as a *table*, is a standard means to document software (Janicki and Wassyng, 2005). This notation can be dated back to *decision tables* and *state transition tables*. In the late 1970s, David Parnas introduced this concept (Alspaugh *et al.*, 1992). Now, tables are widely used in analyzing requirements and documenting specifications for critical software.

In our protocol, we apply tabular expressions as the means to represent the original program. A program verified in our work is expressed as a table graph $G_t$, which is organized by a bundle of tables by certain relations. $G_t$ consists of a table set $T = (T_1, T_2, ..., T_n)$ and a graph structure $G_{struc}$, where $T_i \in T$ is an individual table. If not specifically claimed, $G_t = [T, G_{struc}]$. If an encryption scheme is applied to such a table graph, the encryption version of $G_t$ is $G'_t = [T', G'_{struc}]$.

$$x \rightarrow \begin{array}{|c|c|}\hline \text{Condition} & \text{Function} \\ \hline p_1(x) & f_1(x) \\ \hline p_2(x) & f_2(x) \\ \hline \dots & \dots \\ \hline p_n(x) & f_n(x) \\ \hline \end{array} \rightarrow T(x)$$

Figure 2.1: A single table (from Cai *et al.*, 2016)

Among the variants of tables, we adopt a simple version of tables (Wassyng and Janicki, 2003) with two columns. Observing figure 2.1, a table consist of a *Condition* column and a *Function* column. Given an input $x$, if it satisfies the predicate $p(x)$ in the condition column, $x$ can be computed with the corresponding function $f()$ in the function column.

**Definition 1.** *$T = \{T_1, T_2, ..., T_n\}$ is a set of tables with size $n$. $\forall i \in \{1, ..., n\}, T_i = (r_{1,i}, ...r_{ir,i}, ...r_{rn_i,i})$, where $ir \in \{1, ..., rn_i\}$. $r_i^{ir} = (p_{ir,i}, f_{ir,i})$ is a row of $T_i$ and $rn_i$ denotes the number of rows in table $T_i$. $p_{ir,i}$ is the left hand side or lhs predicate of $r_{ir,i}$ and $f_{ir,i}$ is the right hand side or rhs function of $r_{ir,i}$. $T_i$ accepts an input $x_i$ and computes it row by row. If $p_{ir,i}(x_i) = True$, $x_i$ is computed by $f_{ir,i}$ and $f_{ir,i}(x_i)$ becomes the output of $T_i(x_i)$.*

According to the properties of tabular expressions (Wassyng and Janicki, 2003), the predicates in a table should satisfy the properties of *disjointness* and *completeness*:

**Completeness:** $\forall x, p_1(x) \vee p_2(x) \vee ... \vee p_n(x) = True$

**Disjointness:** $\forall x, p_u(x) \wedge p_v(x) = False, \forall u, v \in [1, n], u \neq v$

In our case, a program is represented as a table graph $G_t = [T, G_{struc}]$. Each individual table $T_i \in T$ is defined as above. $G_{struc}$ denotes the geometric structure of how the tables are organized. In $G_{struc}$, row $r_{ir,i}$ in table $T_i \in T$ represents a vertex, and the connections between two vertices is an edge. The practical meaning of edges is the output of the start row is the input of the destination row. For this reason, $G_{struc}$ is a directed graph. An example of table graph is shown in figure 2.2.

In the table graph $G_t$, a source vertex *Input* denotes where the external inputs come from, and a sink vertex *Output* denotes where the table's results output. For simplicity, we make an assumption:

**Assumption 1.** *The table graph of a program is acyclic.*

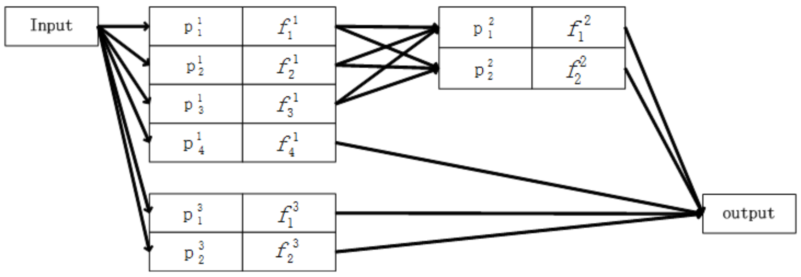


Figure 2.2: A typical table graph (from Cai *et al.*, 2016)

**Definition 2.** $G_{struc} = (V, E)$ *is a rooted directed graph which is called the* structure graph *of* $G_t$. *E is the set of edges, and V is the set of vertices. An edge* $e = (r_{ur,u}, r_{vr,v}), e \in E$ *connects two vertices* $r_{ur,u}, r_{vr,v} \in V$*, where* $r_{ur,u}, r_{vr,v}$ *are the rows in table* $T_u, T_v$*, and* $ur, vr$ *are the row indices.* $e = (r_{ur,u}, r_{vr,v})$ *is directed and starts from the* $u_{th}$ *row of* $T_u$ *to the* $v_{th}$ *row of* $T_v$.

### 2.1.2 Circuits

As garbled circuits is the major technique in our construction, the original table graph should be transformed into a format which the protocol of garbled circuits can process. The format is Boolean circuits. A *Boolean circuit*, or circuit, is a concept derived from electrical engineering, indicating a combination made up of individual electronic components. Applied to computer science, the circuit represents a model of computation (Vollmer, 1999). Each gate in the circuit is a function, computing the most basic Boolean operation, like conjunction, disjunction, and negation (xor may also be included).

In order to use this concept without ambiguity, we should first provide the definition of *what is a circuit*. Our definition is a simplified version derived from the work of (Bellare *et al.*, 2012). An illustrative example of a standard circuit is shown in Figure 2.3.

**Definition 3.** *A circuit $CT$ is a 6-tuple $f = (l_{IN}, l_{OUT}, m, gn, W, G)$. $l_{IN}$ is the length of the input bits and $l_{OUT}$ is the length of output bits. $m$ denotes the number of wires, and it satisfies $m = l_{IN} + gn$, where $gn$ is the number of gates. $W$ is the set of all wires included in $CT$. A wire $w_j \in W$ denotes the $j_{th}$ wire, where $j \in [1, m]$. $w_j^0 = 0, w_j^1 = 1$ is the bit value of $w^j$. $G$ is the set of all gates included in $CT$. A gate $g_i \in G, i \in [1, gn]$, represents a Boolean gate with certain functionality $g : \{0,1\}^2 \to \{0,1\}$ or $\{0,1\} \to \{0,1\}$. The functionality of each gate is basic Boolean gates such as $AND, OR, XOR$ and $NOT$.*

To simplify the discussion, Boolean circuits in our case have the same length of input wires and output wires.

**Assumption 2.** *In a Boolean circuit $CT$ has the same length $l$ of input wires and output wires, i.e. $l_{IN} = l_{OUT} = l$.*

By the theory of computation complexity, the computational power of the circuit model is equivalent to the power of Turing Machine model (Kaye *et al.*, 2007,

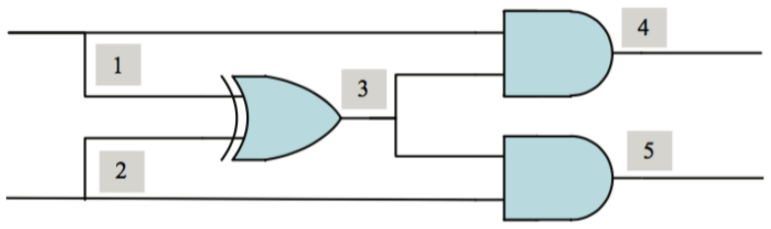


Figure 2.3: A simple circuit with its gates and wires (from Bellare *et al.*, 2012)

Chapter 7). In other words, any computation executed in a Turing Machine can be translated into circuits with the same functionality. In this work, we skip the discussion about how to transform the normal program into Boolean circuits and regard this transformation as a plausible process by default. Based on this fact, we make this assumption:

**Assumption 3.** *A table $T$ as specified in Definition 1 can be transformed into a Boolean circuit $CT$ which obeys Definition 3 and $CT$has the equivalent semantics to $T$.*

Now we define the concept *circuit graph.* Likewise, a circuit graph $G_c$ consists of a circuit set $CT = (CT_1, ..., CT_n)$ and a structure graph $G_{struc}$. The structure graph in $G_c$ is similar to the one in $G_t$, but the concepts like rows, condition columns and result columns are not used. additionally, the vertices in $G_{struc}$ become circuits rather than rows. An example of circuit graph is shown by figure 2.4. Consequently, we present the definition for the circuit graph.

**Definition 4.** *A circuit graph $G_c = [CT, G_{struc}]$ is derived from a table graph $G_t$. $CT_i \in CT$ is the $i_{th}$ circuit transformed from $i_{th}$ table $T_i$, so the input to $T_i$ also works on circuit $CT_i$. An input to $CT_i$ is denoted as $x_i = (x_{1,i}, ...x_{j,i}, ...x_{l_i,i})$, where $l_i$ is the* *input length of* $CT_i$. $x_{j,i} \in x_i$ *is the bit value of* $j_{th}$ *wire in* $CT_i$*'s input and* $j \in [1, l_i]$. *The output of* $CT_i$ $y_i = (y_{1,i}, ...y_{j,i}, ...y_{l_i,i})$ *is also a bit sequence, and here we assume the length of the output is the same as the input. An edge in* $G_c$ *is a tuple* $(u, v)$, $u$ *represents* $CT_u$ *and* $v$ *represents* $CT_v$.

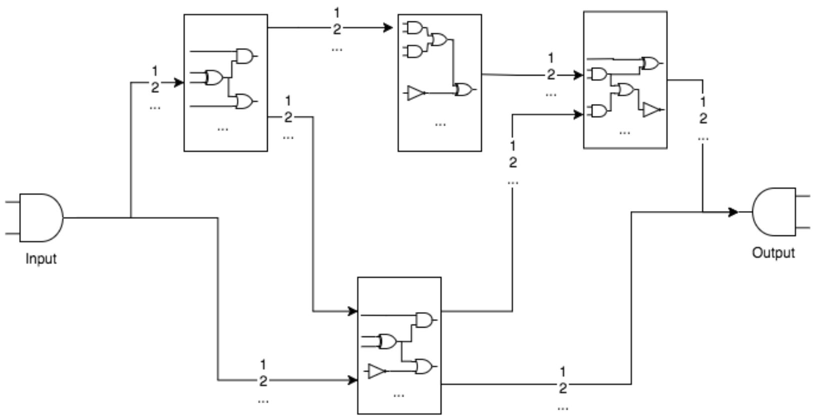


Figure 2.4: A demonstration of a circuit graph

Replacing tabular expressions by circuits to document the program is a necessary step. First, it is a prerequisite of the protocol of garbled circuits (actually $FHE$ is also executed on circuits (Gentry *et al.*, 2009)). Second, the structure of a program can be simplified in a circuit graph. If the program is represented as a table graph $G_t$, the rows represent the vertices in $G_t$. Assumed that $n$ denotes the number of tables in $G_t$ and $rn$ denotes the average number of rows in each table, the number of edges in $G_t$ is estimated at $n \cdot rn$. Representing the program in such a way brings difficulty in encrypting the program directly, so the tables in $G_t$ may be necessarily transformed into single-row tables, like what the work of (Cai *et al.*, 2016) did. This transformation also changes the way to represent functions and introduces some auxiliary symbols, like $\perp$ and $\top$. In this case, the representation of program has been complicated. If the program is represented as a circuit graph $G_c$, the vertices in such a circuit graph are circuits. Without further transformation, the relations between circuits can be maintained, even after encrypting.

Circuit graph $G_c$ does not contain the logic structures like “row” or ”column”, but this change does not affect the functionality of the circuits. Because a row in a table may be just a cluster of gates or a small-size circuit after transformation, while a column is just a logic structure of organizing the rows. Nevertheless, in Boolean representation, the structure of the rows can still be maintained. This job involves a technique called *gate soldering* (Mood *et al.*, 2014; Nielsen and Orlandi, 2009), but we do not adopt it into our construction.

From now on, we always use circuits to represent the program and replace the table graph $G_t$ by a circuit graph $G_c$.

## 2.2 Strategy of Evaluation and Verification

Before proposing the construction of partial white-box verification, a definition about what kind of verification is in discussion should be provided. For ease of explanation, we regard the circuit graph and the specification as functions $G_c$ and $G_{spec}$, and denote the *domains*, all the possible inputs, of $G_c$ and $G_{spec}$ as $D$ or $D_{spec}$. The range, all the possible outputs, of $G_c$ and $G_{spec}$, are denoted as $R$ or $R_{spec}$.

**Evaluation.**

The process of evaluating $G_c$ obeys the *precedence constraints*: In a graph $G = [V, E]$, a edge $e(v_i, v_j)$ implies that the task in vertex $v_i$ must occur before $v_j$, where $e \in E, v_i, v_j \in V$. In our case, if a circuit $CT_i$'s input is the result of $CT_j$'s output, $CT_j$ must be evaluated before evaluating $CT_i$. The Assumption 1 indicates the structure graph of $G_t, G_c$ are directed acyclic graphs ($DAG$). For this reason, we need to assign the vertices of $G_t, G_c$, namely the tables or circuits, by a *topological ordering.*

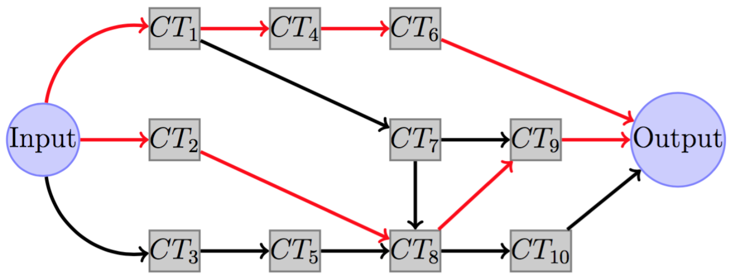
**Definition 5.** *A topological order in a directed graph* $G = (V, E)$ *is an ordering of its vertices as* $v_1, v_2, ..., v_n$ *so that for every edge* $(v_i, v_j)$ *we have* $i < j$*.*

By such a topological order, each circuit $CT_i \in G_c$ is assigned with a unique index. Particularly, the source vertex *Input*, which represents the external inputs to the program, is indexed as $CT_0$; the sink vertex *Output*, which represents the external outputs of $G_c$, is indexed as $CT_{n+1}$.

$G_c$ can accept one or more inputs, and output one or more results. The type of inputs which are incident from vertex *Input* is called as *external inputs*, and we denote the set of actual external inputs (the set may contain multiple inputs, or only one input) which is used to evaluate $G_c$ as $EX$. In contrast, we define the results which are incident to vertex *Output* as *external outputs*, and the set of actual external outputs coming out from $G_c$ as $EY$. According to the types of inputs, the circuits are categorized into three sets. $IC$ represents the set of circuits accepting external inputs and $OC$ represents those end circuits producing external outputs. Those circuits $CT_i$ that are in neither $IC$ and $OC$ are *intermediate circuits*, which are run with the results derived from previous circuits.

Now we can define the concept of evaluation. Focused on an external input $ex \in EX$, the circuits that are passed by during evaluating $G_c$ with $ex$ form a direct path $P = (CT_{head}, ..., CT_{end})$, where $CT_{head} \in IC, CT_{end} \in OC$. By Assumption 1, $P$ is acyclic. Presently we define the strict meaning of evaluation. A demonstration of the evaluation process is shown in Figure 7.

**Definition 6.** *Evaluating* $G_c$ *is a process of repeated composition of executing circuits* $CT_i \in CT$*. For a circuit* $CT_{head} \in IC$*,* $CT_{head}$ *takes as input* $ex \in EX$*. For an* *intermediate circuit $CT_i$ or an end circuit $CT_{end} \in OC$, it takes as input the outputs of previous circuits. A successful path of evaluating $G_c$ is that given an external input $ex \in EX$, the execution starts from the vertex $Input$, ending at the vertex $Output$.*



The paths of red lines are successful processes of evaluation. An input $x$ is evaluated by all connected circuits, only successful evaluation would be colored in red.

Figure 2.5: A demonstration of the evaluation process

**Verification.** After clarifying how to evaluate the circuit graph, we can move forwards to discuss the what type of verification should be here. The first thing is to obtain the test cases. In our settings, we use an algorithm $VGA$ to generate the test cases. $VGA$ takes publicly known $G_{struc}, G_{spec}$, and a security parameter $K$ as the inputs, generating the test cases for $G_{spec}$ and $G_c$ (Hayhurst *et al.*, 2001). $G_{spec}$ denotes the specification described as a table graph. It must be revealed as the reference to check the verification results, which will be used in $VS.Eval$ (Section 3.4.6 and Algorithm 8). For this reason, the specification can be held by the developer, the verifier or a third party.

The verifier executes either the whole system, or a subsystem along a partial path. In the latter way, executing $G_c$ with the test cases may not cover the complete logic of the software system. By a set of known inputs $X$ and a set of known outputs $Y$, the verifier can conduct a partial path taking $X$ as inputs, and verify if the results match the outputs in $Y$. The input-output tuples $(X, Y)$ are *critical points* $(CP)$.

Now we can provide the strategy of verification. With the same test cases generated from $VGA$, the verifier executes the circuit graph $G_c$ and the specifications $G_{spec}$. If the results of $G_c$ and $G_{struc}$ coincides, we can say that $G_c$ passes the verification. The test cases can be replaces by the tuples of $CP$. For $x \in EX$, if $y = G_c(x)$ and $y \in Y$, the verification succeeds on the pairs of $CP$.

A *security parameter* $K$ can be used as a unary parameter in a function, representing $K$ 1's. With the above basis, we define a *trusted verifier* (or *trusted verification algorithm*, derived from (Cai *et al.*, 2016)):

**Definition 7** (Trusted verifier)**.** *A* trusted verifier $V$ *is an algorithm that uses $r$ random bits, and such that*

$$Pr_r\left[ V(1^K, G_c, G_{spec}, CP, VGA)(r) = \begin{cases} 0, & \textit{if } \exists X \in EX\textit{: } G_c(X) \neq G_{spec}(X) \\ 0, & \textit{if } (CP \neq \varnothing) \wedge (\exists (X,Y) \in CP\textit{: } G_c(X) \neq Y) \\ 1, & \textit{if the previous two conditions are not true} \end{cases} \right]$$

$\geq 1 - negl(K)$

The properties of security and correctness can be elicited from the the above definitions about evaluation and verification, while the rigid description is specified in definitions 15 and 16.

- The security requirements contains:
  - Protecting the privacy of contents in each table;
  - Concealing the intermediate values which are the results of executing circuits;

  - Guaranteeing the security of protocol can hold even in the malicious case.

- The correctness requirements contains:
  - The results of executing the circuit graph is the same as the results of executing the specification.
  - A trusted third party can obtain the same results if repeating the verification.

A trusted verifier satisfies Definition 7, as it behaves in the way that a verifier is supposed to behave and does not deviate from the normal procedure. The verification is judged as failure if the results do not coincide with the outputs of $G_{spec}$ or $CP$.

The problem in this work (or in the real-life situation) is that the verifier can only get access to an *encrypted* program, namely an encrypted circuit graph $G'_c$. Thus, we need to design a protocol to realize the verification executing on the encrypted circuit graph. Throughout this thesis, our work is around this core question.

## 2.3 Pseudorandom Generators

A pseudorandom generator refers to a deterministic algorithm that takes a short, truly random string as a seed and expands the short string into a much longer string that appears to "random" (Goldreich, 2006, p. 3). Many encryption schemes are built upon pseudorandom generators and play a fundamental role in cryptography.

Pseudorandom generators rely on an important assumption: *the existence of one-way functions can lead to the existence of pseudorandom generators (Katz and Lindell, 2007).* The pseudorandom generator is an important component in our whole cryptographic construction. Here we borrow the authoritative explanation from (Goldreich, 2006, Chapter 3) and present the definition for pseudorandom generators.

**Definition 8.** *$p(.)$ is a function $\mathbb{N} \to \mathbb{N}$, representing the length of a string with a parameter. For any input $s \in \{0,1\}^n$, $n$ is the length of seed $s$. $G$, a deterministic polynomial-time algorithm, is a pseudorandom generator that two following conditions hold:*

1. *For all $n \in \mathbb{N}$, $G$ expands $s$ and satisfies $G(s) = p(s)$.*
2. *For a probabilistic polynomial time (or "p.p.t." for short) distinguisher D, there exists a negligible function $negl(.)$ such that:*

$$|Pr[D(r) = 1] - Pr[D(G(s)) = 1]| \leq negl(n) \tag{2.1}$$

*where $r$ is chosen uniformly at random from $\{0,1\}^{p(n)}$, the seed $s$ is chosen uniformly at random from $\{0,1\}^n$.*

## 2.4 Garbled Circuits Protocol

In this section, we present a brief description about the protocol of Yao garbled circuits. Secure function evaluation ($SFE$) indicates a problem about how two parties can compute a function without leaking their private inputs to the opponent. To solve this problem, Andrew Yao proposed garbled circuits (Yao, 1982, 1986). The original garbled circuits for $SFE$ did not relate to protecting the content of the circuit. The following work, like (Abadi and Feigenbaum, 1990) extends the security of garbled circuits to guarantee function privacy.

Following the work in (Goldwasser *et al.*, 2013), we provide the definition of garbled circuits such that:

## 2.4.1 Defining Garbled Circuits

**Definition 9.** *$C$ is a circuit with $n$ bits of input and output. A scheme for garbled circuits protocol is a tuple of p.p.t. algorithms GB=(GB.Garble,GB.Enc,GB.Eval) as following:*

1. *$GB.Garble(1^K, C)$ takes as input the security parameter $K$ and a circuit $C$, outputting a garbled circuit $\Gamma$ and a series of garbled keys $gk = \{k_j^0, k_j^1\}_{j=1}^m$ for all wires in $C$, where $j$ denotes the index of wires and $m$ is the total number of wires in $C$.*

2. *$GB.Enc(gk, x)$ takes as input the value $x = (x_1, x_2, ..., x_n), x_j \in \{0, 1\}$ and garbled keys $gk$ for the circuit $C$, outputting an encoded $x' = (k^{x_1}, k^{x_2}, ..., k^{x_n})$, where $k^{x_j}$ indicates the corresponding garbled key of bit $x_j$.*

3. *$GB.Dec(gk, y')$ takes as input $gk$ and $y'$, and outputs a decrypted value $y = (y_1, y_2, ..., y_n), y_j \in \{0, 1\}$.*

4. *$GB.Eval(\Gamma, x')$ takes as input a garbled circuit $\Gamma$ and an encoded value $x'$, outputting $y' = (k^{y_1}, k^{y_2}, ..., k^{y_l})$ which is an encrypted value of $y = C(x)$.*

**Correctness:** *For a polynomial length $n$, a sufficiently large $K$, a circuit $C$, and all $x \in \{0, 1\}^n$,*

$$Pr[(\Gamma, gk) \leftarrow GB.Garble(1^K, C); x' \leftarrow GB.Enc(gk, x);$$

$$y' \leftarrow GB.Eval(\Gamma, x'); y \leftarrow GB.Dec(gk, y') : C(x) = y] = 1 - negl(K)$$

**Security:** *It is assumed that there exists a p.p.t. simulator $S_{GB}$ and a p.p.t. adversary $A$, distinguisher $D$. For a garbled circuits scheme $GB$, its security guarantees the privacy of the input $x$ and the circuit $C$. For a sufficiently large security parameter $K$:*

$$\begin{aligned}|Pr[(x, C, \alpha) \leftarrow A(1^K); (\Gamma, gk) \leftarrow GB.Garble(1^K, C);\\ x' \leftarrow GB.Enc(gk, x) : D(\alpha, x, C, \Gamma, x') = 1]\\ -Pr[(x, C, \alpha) \leftarrow A(1^K);\\ (\tilde{\Gamma}, \tilde{x'}) \leftarrow S_{GB}(1^K, C(x), 1^{|C|}, 1^{|x|}) : D(\alpha, x, C, \tilde{\Gamma}, \tilde{x'}) = 1]|\\ = negl(K),\end{aligned}$$

*where $\alpha$ represents any state that $A$ may want to convey to $D$. $S_{GB}$ representing a simulator of garbled circuits protocol is able to simulate the view of the real protocol only by the result of computing $C$, and the size of the circuit and input.*

### 2.4.2 Description of Garbled Circuits

In this section provides a brief explanation about how the protocol of garbled circuit, $GB$ works. Here we focus on the execution on a single circuit. The involved two parties are the verifier and the developer. The protocol can be divided into five stages (Snyder, 2014).

**Step 0. Transforming the table into a circuit**

Originally, the program is in the form of tables and owned by the developer. Thus, the developer should transform the tables into a Boolean circuits $C$ so that $GB$ can process the program.

**Step 1. Garbling the circuit**

This step describes how *GB.Garble* works and garbles a circuit. The circuit $C$ consists of many gates and wires, which are organized in an intricate structure (Section 2.1.2). Garbling a circuit is actually to garble the gates and wires within the circuit.

- *Garbling a wire.* By the requirement of input privacy, the bits for wires should be hidden from the verifier. Here $GB$ uses pseudorandom generator to encrypt each wire.

  Produced from a pseudorandom generator $G$ on a seed $s$, two pseudorandom strings $v_j^0, v_j^1$ correspond to the bit 0 and 1 for a wire $w_j$. With an extra random binary permutation bit $p_j$, the garbled keys for a wire are created as $k_j^0 = (v_j^0 \parallel (0 \oplus p_j))$ and $k_j^1 = (v_j^1 \parallel (1 \oplus p_j))$ (Malkhi *et al.*, 2004), where symbol $\parallel$ represents the operation of concatenation, and $\oplus$ represents the exclusive or.

- *Garbling a gate.* The content of the circuit $C$ should also be hidden from the verifier. As the functionality of a circuit is organized by the gates, what $GB$ encrypts is each gate.

  Any kind of gates can be expressed as truth tables. Considering a typical gate $g$ with two input wires and one output wire, its truth table has four entries which can be represented as $A_g^{0,0}, A_g^{0,1}, A_g^{1,0}, A_g^{1,1}$, namely the results of four possible input tuples: $(X = 0, Y = 0), (b_X = 1, b_Y = 0), (b_X = 0, b_Y = 1), (b_X = 1, b_Y = 1)$, where $X, Y$ denote the two input wires.

  Garbling a gate is to garble the entries of the corresponding truth table. Prior to that, the wires of the gate (two inputs wires and an output wire) should be encoded into garbled keys as the last step. In the garbled truth table, four entries are denoted as $c^{0,0}, c^{0,1}, c^{1,0}, c^{1,1}$. Using $SHA-1$ (Huang *et al.*, 2011), the garbled entries are produced by

$$c^{X,Y} = \text{SHA-1}(k_X^{b_X} \| k_Y^{b_Y} \| Z) \oplus k_Z^{g(b_X,b_Y)}, \tag{2.2}$$

where $Z$ represents the output wire. Regarding $g$ as a function, the bit values of $X, Y, Z$ should be in such a relation

$$b_Z = g(b_X, b_Y). \tag{2.3}$$

Overall, a gate is replaced by the four garbled entries

$$c^{0,0} = \text{SHA-1}(k_X^0 \| k_Y^0 \| Z) \oplus k_Z^{g(0,0)}$$
$$c^{0,1} = \text{SHA-1}(k_X^0 \| k_Y^1 \| Z) \oplus k_Z^{g(0,1)}$$
$$c^{1,0} = \text{SHA-1}(k_X^1 \| k_Y^0 \| Z) \oplus k_Z^{g(1,0)}$$
$$c^{1,1} = \text{SHA-1}(k_X^1 \| k_Y^1 \| Z) \oplus k_Z^{g(1,1)}$$

Importantly, this process can be spread over multiple gates. Just like what the gates are computed in real circuit, the computation on a garbled circuit can be passed through the garbled truth tables. The garbled keys in a garbled truth table can appear in another garbled truth table, which carries the results of the previous gate forward into the next gates. A demonstration about this process is shown in Figure 2.6. The garbled keys of the output wire $w_3$ in gate $g_1$ (Figure 2.7(b)) appear in the garbled truth table of gate $g_3$ (Figure 2.8(b)).

Dealing with all wires and gates in circuit $C$ as the above construction, the job of building a garbled circuit is finished, and a garbled circuit $\Gamma$, as well as the garbled keys $\{k_j^0, k_j^1\}_{j=1}^m$, is generated.

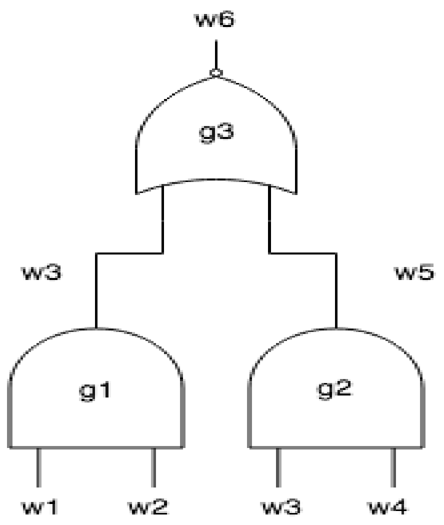


Figure 2.6: An example of multiple gates (from Snyder, 2014)

| $w_1$ | $w_2$ | $w_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a) Truth table for a AND gate $g1$

| $w_1$ | $w_2$ | $w_3$ |
|---|---|---|
| $k_1^0$ | $k_2^0$ | $k_3^0$ |
| $k_1^0$ | $k_2^1$ | $k_3^0$ |
| $k_1^1$ | $k_2^0$ | $k_3^0$ |
| $k_1^1$ | $k_2^1$ | $k_3^1$ |

(b) Garbled truth table for gate $g_1$

Figure 2.7: The truth table and the garbled truth table for Gate $g_1$

| $w_3$ | $w_5$ | $w_6$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(a) Truth table for a OR gate $g_3$

| $w_3$ | $w_5$ | $w_6$ |
|---|---|---|
| $k_3^0$ | $k_5^0$ | $k_6^0$ |
| $k_3^0$ | $k_5^1$ | $k_6^1$ |
| $k_3^1$ | $k_5^0$ | $k_6^1$ |
| $k_3^1$ | $k_5^1$ | $k_6^1$ |

(b) Garbled truth table for gate $g3$

Figure 2.8: The truth table and the garbled truth table for Gate $g_3$

It is important to note, the developer knows the mapping tuples $map = \left([k_j^0, 0], [k_j^1, 1]\right)_{j=1}^{m}$ between garbled keys $(k_j^0, k_j^1)$ and its corresponding real values. The mapping tuples can be used to decrypt the outputs of evaluating the garbled circuit $\Gamma$. The garbled keys produced in this phase will be used to encode the input in next step.

**Step 2. Encoding the inputs**

This step describes how to use garbled keys to encode the given input $x$. The core of this job is the protocol of *oblivious transfer*. An oblivious transfer protocol (or *OT* for short) is a technique allowing two parties, a client and a receiver, to exchange the requested value without leaking other unauthorized information. The principles of *OT* is provided in Section 2.5.

At the beginning, the developer owns the garbled keys $\{k_j^0, k_j^1\}_{j=1}^n, j \in [1, n]$. Via *1-out-of-2 OT*, the verifier exchanges each bit $x_j \in x = (x_1, ...x_n)$ for the corresponding garbled keys $k^{x_j}$. As specified in Algorithm 1, the verifier sends a set of public keys $(k_{0,j}^{pub}, k_{1,j}^{pub})_{j=1}^n$ to the developer, rather than the original input bits $x = (x_1, ...x_n)$. Then, the developer uses these public keys to encrypt the garbled keys $\{k_j^0, k_j^1\}_{j=1}^n$ at hand, returning the pairs of cipher $\{c_{0,j}, c_{1,j}\}_{j=1}^n$. Particularly, in a pair of cipher $(c_{0,j}, c_{1,j})$, the verifier can only decrypt one piece of them, and $x_j$ decides which one can be decrypted. If $x_j = 0$, $c_{0,j}$ can be decrypted; if $x_j = 1$, $c_{1,j}$ can be decrypted. Consequently, the verifier only receives the corresponding garbled keys $\{k^{x_j}\}_{j=1}^n$. In the whole procedure, the developer does not know the values $\{x_j\}_{j=1}^n, x_j \in x$, and the verifier cannot acquire the other garbled keys $\{k^{1-x_j}\}_{j=1}^n$. The procedure can be illustrated in Figure 2.9.

In the above discussion, only the verifier has the input to participate the computation. If the developer also has inputs, he can encode the inputs by himself before encoding the verifier's inputs. Becasue the developer generates the garbled circuit, he owns the mapping tuples $map = ([k_j^0, 0], [k_j^1, 1])_{j=1}^m$. For an input $\dot{x}$ from the developer, the developer can replace each bit of $\dot{x}$ with the corresponding garbled key by

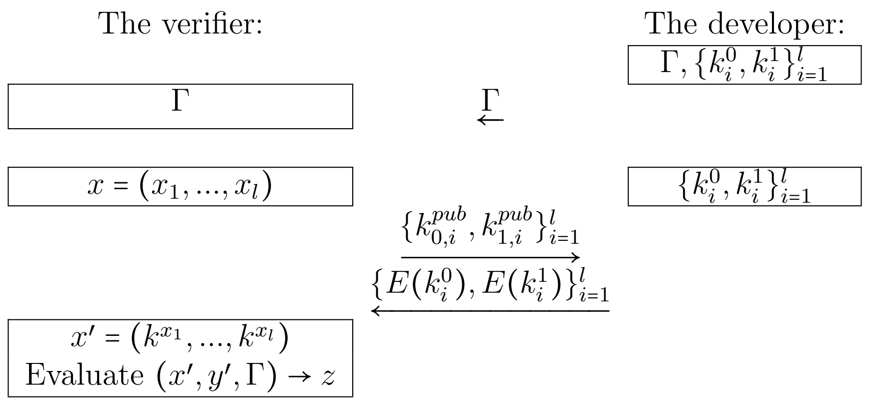

The verifier:
The developer:
$\Gamma, \{k_i^0, k_i^1\}_{i=1}^l$
$\Gamma$
$\xleftarrow{\Gamma}$
$x = (x_1, ..., x_l)$
$\{k_i^0, k_i^1\}_{i=1}^l$
$\xrightarrow{\{k_{0,i}^{pub}, k_{1,i}^{pub}\}_{i=1}^l}$
$\xleftarrow{\{E(k_i^0), E(k_i^1)\}_{i=1}^l}$
$x' = (k^{x_1}, ..., k^{x_l})$
Evaluate $(x', y', \Gamma) \to z$

Figure 2.9: The protocol of garbled circuits

the mapping tuples $map = ([k_j^0, 0], [k_j^1, 1])_{j=1}^m$. Then, the developer sends the encoded $\bar{x}'$ to the verifier as well as the garbled circuit. As the verifier does not have the mapping tuples $map$, the real bits behind $\bar{x}'$ is blind to the verifier.

**Step 3. Computing the garbled circuit**

This step corresponds to *GB.Eval*. Reversing the Equation (2.2), the garbled keys for the output wire can be computed by the following equation:

$$k_Z^{g(b_X, b_Y)} = \text{SHA-1}(k_X^{b_X} \| k_Y^{b_Y} \| Z) \oplus c^{X,Y}. \tag{2.4}$$

Equation (2.4) indicates that, with the complete set of garbled entries $C^{X,Y}$ for a gate and two garbled keys for two input wires respectively, the garbled key for output wire $Z$ can be computed.

Take the gate $g_1$ in Figure 2.7(b) as example. Assuming that $X$ is the value of wire $w_1$ and $Y$ is the value of wire $w_2$, if $X = 0$ and $Y = 1$, the garbled keys for the two wires are $k_1^0, k_2^1$. In **Step 1**, the verifier obtains the garbled entries for gate $g_1$, $c^{0,0}, c^{1,0}, c^{0,1}, c^{1,1}$. Taking as input the garbled entries and garbled keys for each wire, the verifier can compute Equation (2.4). Particularly, only the correct garbled entry can work out. By Equation (2.3) and Figure 2.7(a), $Z = g(0,1) = 0$. With garbled keys $k_1^0, k_2^1$ at hand, the verifier acquires $k_3^0$ as the result of computing gate $g_1$. If computing the garbled entries with incorrect garbled keys, we use a symbol $\perp$ to represent the result.

Given the input $x = (k^{x_1}, k^{x_2}, ..., k^{x_n})$, the above computation can be spread over all the garbled truth tables in $\Gamma$ until the process reaches the the output gates. Eventually, an output $y' = (k^{y_1}, k^{y_2}, ..., k^{y_n})$, which consists of a sequence consisting of garbled keys, comes out from garbled circuit $\Gamma$.

**Step 4. Decrypting the results**

Lastly, the verifier needs a decrypted output. There are two ways to decrypt $y'$ (Lindell and Pinkas, 2007). The first one is that the developer uses the mapping tuples $map = ([k_j^0, 0], [k_j^1, 1])_{j=1}^m$ to decrypt $y' = (k^{y_1}, k^{y_2}, ..., k^{y_n})$. Each garbled key $k^{y_j}$ can always find a tuple $[k_j^0, 0], [k_j^1, 1]$ and acquire the corresponding bit. Then the developer returns the sequence of decrypted bits to the verifier. This job denotes to *GB.Dec*. The another way is that the end gates of $\Gamma$ directly output the decrypted values directly. If the garbled keys of output wires is replaced by the corresponding bits (see Figure 2.1), the results of the garbled truth tables become real bits. In this method, decrypting the outputs is integrated into the circuits and *GB.Dec* is not necessary to invoke. Both of the two ways still protect the privacy of the circuit. In our settings, we choose the second way and thus *GB.Dec* is omitted in the scheme of garbled circuits.

| $w_1$ | $w_2$ | $w_3$ |
|---|---|---|
| $k_1^0$ | $k_2^1$ | 0 |
| $k_1^1$ | $k_2^0$ | 0 |
| $k_1^0$ | $k_2^1$ | 0 |
| $k_1^1$ | $k_2^1$ | 1 |

Table 2.1: A garbled truth table outputs a real bit directly

## 2.5 Oblivious Transfer Protocol

An oblivious transfer protocol ($OT$) is a building block in garbled circuits. Its functionality can be described as (Goldreich, 2004, Section 7.3.2)

$$(i, (s_1, .., s_k)) \mapsto (s_i, \lambda). \tag{2.5}$$

Equation (2.5) means that, in a 1-out-of-k OT, one party (the sender) can request a value $s_i$ by sending an index $i$ to another party (the receiver), who holds $k$ values $s_1, ..., s_k$. The symbol $\lambda$ indicates that the $OT$ returns nothing to the receiver at the end of the protocol. An $OT$ guarantees that the sender cannot know the remaining values $s_1, ..., s_k$ except $s_i$, and the receiver cannot know the index.

What we interest is the 1-out-of-2 $OT$. The two garbled keys $k^0, k^1$ owned by the developer are requested by the verifier. The verifier sends the input bit, $x_j$, as the index to request the corresponding garbled key. If the input $x = (x_1, ..., x_n)$, there are $|n|$ rounds of $OT$ to exchange the garbled keys. A brief algormic description for 1-out-of-2 $OT$ (Snyder, 2014) is shown in Algorithm 1, which is secure in the semi-honest case. To present the concept of $OT$ more clear, we provide an example in Figure 2.10.

Particularly, we regard the oblivious transfer as a building block in our construction. So the security of the protocol of garbled circuits, and even our protocol, can be reduced to the security of $OT$.

**Definition 10** (Security of Oblivious Transfer)**.**

*For the sender:* $View_{sender}(i, \{s_0, s_1\}) \approx S_{OT}(i, s_i)$

*For the receiver:* $View_{receiver}(i, \{s_0, s_1\}) \approx S_{OT}(\{s_0, s_1\}, \lambda)$

*$S_{OT}$ is the simulator who manages to generate the indistinguishable views of the sender and the receiver. Operation $\approx$ represents a computational indistinguishable relation between two distribution ensembles.*

**Algorithm 1** Semi-honest 1-out-of-2 Oblivious Transfer.
$E = (E.Enc, E.Dec)$ is a public-key encryption scheme.

```
1: The receiver has a set of strings $\{s_{0,i}, s_{1,i}\}_{i=1}^{n}$.
2: The sender has a bit sequence $x = (x_1, ..., x_i, ..., x_n)$, where $x_i \in [0,1]$ and $i \in [1,n]$.
   Each bit $x_i$ corresponds to an expected string. If $x_i = 1$, the sender expects $s_{1,i}$;
   if $x_i = 0$, the sender expects $s_{0,i}$.
3: for all $i \in [1,n]$ do
4:     The sender generates a public/private key pair $(pk_i, sk_i)$ along with an auxiliary value $pk'_i$ that is computationally indistinguishable with from $pk_i$. But the sender does not have the private key that $pk'_i$ corresponds to.
5:     The sender then advertises $(k_{0,i}^{pub}, k_{1,i}^{pub})$ as public keys. If $x_i = 0$, $k_{0,i}^{pub} = pk_i$, while $k_{1,i}^{pub} = pk'_i$; if $x_i = 1$, $k_{0,i}^{pub} = pk'_i$, $k_{1,i}^{pub} = pk_i$.
6:     The receiver generates $c_{0,i} = E.Enc(s_0, k_{0,i}^{pub})$ and $c_1 = E.Enc(s_1, k_{1,i}^{pub})$, and sends $c_{0,i}, c_{1,i}$ to the sender.
7:     The sender decrypts $c_{0,i}, c_{1,i}$ and obtains $s_{b,i} = E.Dec(c_{b,i}, sk_i)$, where $b$ is the value of $x_i$.
8: end for
```

| | | | | | | |
|---|---|---|---|---|---|---|
| $w_1$ | 0 | $k^{pub}_{0,1}$ | $k^{pub}_{1,1}$ | $\longrightarrow$ | $k^0_1$ | $k^1_1$ |
| $w_2$ | 1 | $k^{pub}_{0,2}$ | $k^{pub}_{1,2}$ | $\longrightarrow$ | $k^0_2$ | $k^1_2$ |
| $w_3$ | 1 | $k^{pub}_{0,3}$ | $k^{pub}_{1,3}$ | $\longrightarrow$ | $k^0_3$ | $k^1_3$ |
| $w_4$ | 0 | $k^{pub}_{0,4}$ | $k^{pub}_{1,4}$ | $\longrightarrow$ | $k^0_4$ | $k^1_4$ |

| | | | | | | |
|---|---|---|---|---|---|---|
| $w_1$ | 0 | $k^{sk}_{0,1}$ | $\perp$ | $\longleftarrow$ | $E_{k^{pub}_{0,1}}(k^0_1)$ | $E_{k^{pub}_{1,1}}(k^1_1)$ |
| $w_2$ | 1 | $\perp$ | $k^{sk}_{1,2}$ | $\longleftarrow$ | $E_{k^{pub}_{0,2}}(k^0_2)$ | $E_{k^{pub}_{1,2}}(k^1_2)$ |
| $w_3$ | 1 | $\perp$ | $k^{sk}_{1,3}$ | $\longleftarrow$ | $E_{k^{pub}_{0,3}}(k^0_3)$ | $E_{k^{pub}_{1,3}}(k^1_3)$ |
| $w_4$ | 0 | $k^{sk}_{0,4}$ | $\perp$ | $\longleftarrow$ | $E_{k^{pub}_{0,4}}(k^0_4)$ | $E_{k^{pub}_{1,4}}(k^1_4)$ |

Note: Suppose $x = (0110)$. $\perp$ indicates a meaningless string. The blue cells in the first table represent which bits' garbled keys the sender wants to obtain; the blue cells in the second table are the ones that the sender can decrypt.

Figure 2.10: An example of how *OT* exchanges bits for garbled keys

In this work, *OT* should be secure in malicious cases. Fortunately, the work in (Bellare and Micali, 1989; Naor and Pinkas, 2001, 2005) has proposed the stronger *OT* which is secure against malicious behavior. So we use this fact in following construction.

## 2.6 Cut-and-choose Strategy

The cut-and-choose approach is a strategy that can enhance the security of the protocol of garbled circuits and address the malicious behavior. Compared to the *zero-knowledge proofs* approach (Goldreich *et al.*, 1987), the cut-and-choose strategy is more plausible and thus it receives more attention (Lindell and Pinkas, 2007; Shen *et al.*, 2011; Mohassel and Riva, 2013; Nielsen and Orlandi, 2009). The core idea behind this method is that each original circuit will have more than one garbled circuits and sets of garbled keys. Figure 2.11 is an example to depict the cut-and-choose strategy. Among the multiple garbled circuits for one circuit $C$, some of them can be used for consistency check, while the rest of garbled circuits are used for being evaluated. By this feature, if the garbled circuits are built incorrectly, the garbled keys are altered, or the results from evaluated circuits are not consistent, the protocol can detect such behavior with high probability. Here we introduce a new security parameter $s$, which is used to indicate how many garbled circuits are generated for a single original circuit.

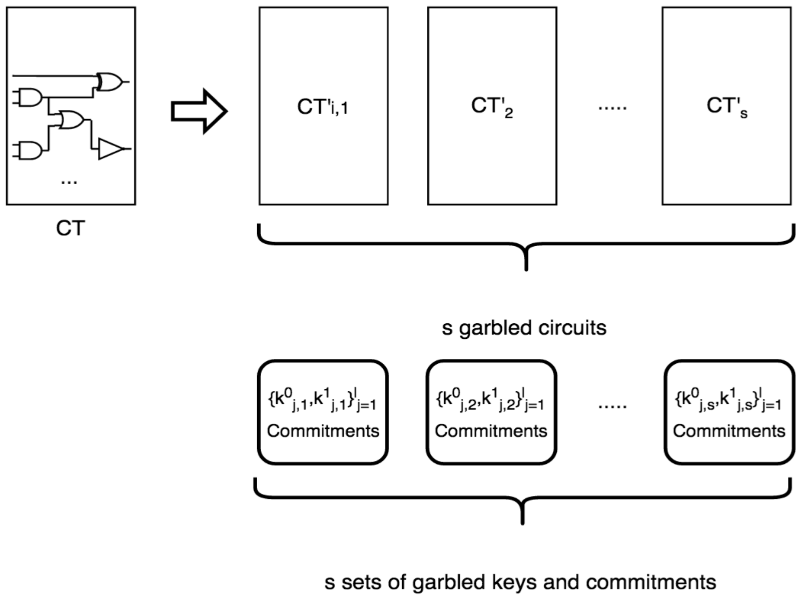

CT
CT'i,1
CT'2
.....
CT's
s garbled circuits
{k0j,1,k1j,1}lj=1 Commitments
{k0j,2,k1j,2}lj=1 Commitments
{k0j,s,k1j,s}lj=1 Commitments
s sets of garbled keys and commitments

Figure 2.11: A demonstration of applying the cut-and-choose strategy

The cut-and-choose approach has evolved into many variants. In this work we adopt the framework in (Lindell and Pinkas, 2007), but some changes have to be made so as to adjust it to our problems. First, Lindell's strategy is designed for the case which both the developer and the verifier provide inputs. However, the inputs only come from one party (the verifier) in our case. On the other hand, in order to check if the garbled circuits are built correctly, Lindell's construction allows the developer to make the content of circuits public. This setting conflicts with the core requirement of the circuit security and it must be removed in our construction.

Besides duplicating garbled circuits in multiple copies, the cut-and-choose approach also involves other techniques to defend malicious behavior. The *commitment scheme* is an efficient two-party protocol which is used for checking data integrity (Goldreich, 2006, Section 4.4.1). Commitment schemes are important components in many cryptography protocols. In our protocol, we apply the commitment scheme to prevent the garbled keys from being altered. For example, in **Step 2** of the protocol of garbled circuits, the verifier sends an input $x_i$ and requests the developer to encode it. It is possible that, the developer uses different garbled keys to encode $x_i$, and thus breaches the security and correctness. In this work, we adopt the *bit-commitment* scheme as the commitment scheme (Section 2.7).

An important issue about the cut-and-choose approach is the overhead. Concerning the $s$ versions of garbled circuits, the protocol generates $ns$ tuples of commitments, $com(k^0_{i,j,r}), com(k^1_{i,j,r})$, where $i, j, r$ are the indices of circuits, wires, and garbled copies. In each circuit $CT_i$, the commitments for the inputs are $2 \cdot (|s \text{ garbled copies}|) \cdot (|\text{wires}|) = 2 \cdot s \cdot n = 2sn$. As $n$ is usually fixed, the computation overhead often depends on the parameter $s$.

## 2.7 Commitment Schemes

In this section we introduce the concept of the bit-commitment scheme. In this work, we adopt the construction from (Naor, 1991) and present the *commitment scheme for* *a single bit* and the *commitment scheme for many bits.* The former is used in coin-tossing protocol (Section 2.8) and the latter is used in the cut-and-choose approach.

The protocol of the bit-commitment scheme has two stages, the *commit stage* and the *reveal stage.* At the commit stage, $P_1$ commits to a piece of information and sends the commitments for the information to $P_2$. In the reveal stage, the commitment can be "opened" and the $P_2$ can check if the information from the "opened" commitment is consistent with the expected information. In our case, the committed information indicates the garbled keys.

**Definition 11** (A commitment scheme for a single bit)**.** *It is assumed that $G()$ is a pseudorandom generator satisfying the definition 8. $G_i(s)$ denotes the first $i$ bits of the pseudorandom sequence on seed $s \in \{0,1\}^n$. $B_i(s)$ denotes the $i_{th}$ bit of the pseudorandom sequence on seed $s$. The construction of the commitment scheme for a single bit is as following:*

- *The commit stage–*

  1. *$P_2$ sets a random vector $\vec{R}$=($r_1, ...r_i, ..., r_{3n}$) where $r_i \in \{0,1\}$ for $1 \leq i \leq 3n$ and sends $\vec{R}$ to $P_1$.*
  2. *$P_1$ selects a seed $s \in \{0,1\}^n$ and sends to $P_2$ the vector $\vec{D} = (d_1, d_2, ..., d_{3n})$ where*

$$d_i = \begin{cases} G_i(s) \text{ if } r_i = 0 \\ G_i(s) \oplus b \text{ if } r_i = 1 \end{cases} \tag{2.6}$$

- *The reveal stage–*

*$P_1$ sends $s$ and $P_2$ verifies that:*

$$\begin{cases} \text{if } r_i = 0, \text{check if } d_i = B_i(s) \\ \text{if } r_i = 1, \text{check if } d_i = B_i \oplus G_i(s) \end{cases} \tag{2.7}$$

A more general case is to commit a long message rather than a single bit. Sticking to one pseudorandom sequence, we can propose a commitment scheme for many bits:

**Definition 12** (A commitment scheme for many bits)**.** *Based on Definition 11, party $P_1$ commits $M = (b_1, b_2, .., b_m)$ to $P_2$.*

- *The commit stage–*

  1. *$P_2$ selects a random vector $\vec{R}$=($r_1, ...r_i, ..., r_{2q}$) where $r_i \in \{0, 1\}$ and $|q|$ of the $r_i$'s are 1. $P_2$ sends $\vec{R}$ to $P_1$.*
  2. *$P_1$ computes a hamming code $c$ for message $M$, where $c \in \{0, 1\}^q$. $c = Hm(b_1, b_2, ..., b_m)$ and the hamming distance between any $c_i, c_{i+1} \in c$ is at least $\epsilon \cdot q$. $P_1$ chooses a seed $s \in \{0, 1\}^n$ and makes a commitment $M' = (b'_1, ..., b'_q)$:*

  $$\begin{cases} \text{if } r_i = 0, b'_i = B_i(s) \\ \text{if } r_i = 1, b'_i = c_i \oplus B_i(s) \end{cases} \tag{2.8}$$

  *$P_1$ sends $M'$ to $P_2$. The commit stage ends.*

- *The reveal stage – $P_1$ sends $s$ and $M$ to $P_2$ and then $P_2$ verifies the correctness of $M'$. To distinguish from the values in the commit stage, $G'(s)$ and $c'$ represent the new pseudorandom sequence and hamming code. $P_2$ conducts such check:*

$$\begin{cases} \text{if } r_i = 0, \text{check if } b'_i = G'_i(s) \\ \text{if } r_i = 1, \text{check if } b'_i = c'_i \oplus G'_i(s) \end{cases} \tag{2.9}$$

**Note:** *Enlarging the Hamming distance between two different original strings reduces the probability of conflicts when generating pseudorandom strings. So we need an efficiently computable function* $Hm : \{0,1\}^m \mapsto C$*, where* $C$ *is a string with satisfactory hamming distance.*

## 2.8 Coin-tossing Protocol

In the cut-and-choose approach (Section 2.6), each circuit has multiple garbled circuits and commitments which can be used for checking or evaluating. Thus, a mechanism that decides which circuits are used for checking or evaluating should be designed. Particularly, selecting which circuits should satisfy fairness and randomness. To achieve this task, we use the *coin-tossing protocol* (Goldreich, 2004, Section 7.4.3).

The coin-tossing protocol is to help two parties agree on a random string which is unbiased towards both of them. In our problems, the developer and the verifier are mutual distrustful with each other. To make an agreement on choosing which circuits are used for checking or evaluating, an efficient way is that both the two parties decide a string cooperatively. The idea of coin-tossing protocol to decide a random bit agreed by the two parties is to force each of the two parties to contribute a random bit, and reliably compute the shared random bit.

**Definition 13.** *A coin-tossing protocol is a two-party protocol for securely computing the random function* $(1^n, 1^n) \mapsto (b, b)$*, where* $b$ *is uniformly distributed in* $\{0,1\}$*.*

For a random vector $r$ and the commitment $C_r(b)$ with $r$ for a bit $b$,

The coin-tossing protocol:

1. Both parties take as input security parameter $1^n$.
2. $P_1$ randomly selects a bit $\sigma \in \{0,1\}$ and $s \in \{0,1\}^n$, and sends c=$C_s(\sigma)$ to $P_2$.
3. $P_2$ randomly selects $\sigma' \in \{0,1\}$ and sends $\sigma'$ to $P_1$.
4. $P_1$ receives $\sigma'$ and output b=$\sigma \oplus \sigma'$, and sends $\sigma, s$ to $P_2$.
5. $P_2$ receives $\sigma, c$ and checks if the c he received in Step 1 is equal to $C_s(\sigma)$. If the equality holds, return $\sigma \oplus \sigma'$; if not, return $\perp$.

# Chapter 3

# A Protocol for Secure and Trusted Partial White-box Verification

This chapter we introduce the protocol for the secure and trusted partial white-box verification.

## 3.1 Problem Analysis

In this section, we dig deeper to analyse the problems. The analysis involves how to use the preliminaries, introduced in Chapter 2, to design the protocol.

**Means of describing the program.** As a default setting, the original program is described by tabular expressions. The software program consists of components, which are implemented by the developer (components may come from different parties, but in our situation, we regard all of the developers as one party). Using tables to describe these components, the orignal program becomes a table graph $G_t = [T, G_{struc}]$, where $T_i \in T = (T_1, ..., T_n)$ denotes each component described by tabular expressions (Section 2.1).

In order to apply garbled circuits, tables in $G_c$ should be transformed into circuits. By the assumption 3, this transformation is an automatic step before we run the protocol. Thus, a table graph $G_t$ is transformed into a directed circuit graph $G_c = [CT, G_{struc}]$, where $CT_i \in CT = (CT_1, ..., CT_n)$ denotes the circuit transformed from table $T_i$.

**Executing the circuit graph.** Evaluation is a process of computing a program with inputs. The Definition 6 specifies the notion of how to execute the program on a circuit graph. The evaluation is conducted by the verifier. Given a set of external inputs $x_i \in EX, x_i = (x_{1,i}, ..., x_{n,i})$, the process of evaluating the original circuit graph $G_c$ can be specified as following steps:

1. $x_i \in EX$ is the input for circuit $CT_i \in IC$, where $IC$ represents the set of circuits accepting external inputs. Computing $x_i$ with the corresponding circuits, the verifier obtains the results $y_i$ for $CT_i$.

2. Except circuits $CT_i \in IC$, the inputs of the remaining circuits are the results of previous computation. Executed like iterative function, the process continues until the end circuits output the results externally. The external outputs are the final result of $G_c$

**Secure verification.** As introduced in Section 1.1, the goal of verification is to check if the program coincides with the specification. The goal is realized by evaluating the program, i.e. the circuit graph $G_c$ and the specification with the same test cases and checking if the results are equal (Section 2.2).

In the meantime, the process of verification must satisfy the security. A trusted verifier, which is defined in Definition 7, is asked to guarantee that the function privacy of the developer is not violated. For this reason, the process of evaluation cannot be performed publicly.

As garbled circuits has the property of function privacy, we apply the technique in the process of evaluation to protect the function privacy during executing $G_c$. Before the process of evaluating $G_c$ starts, the developer calls *GB.Garble* (Definition 9) to garble every circuit $CT_i$, producing the corresponding garbled circuits and garbled keys for all the wires in $CT_i$. This measurement can manage to hide the functionality of each circuit as all the bits are replaced by pseudorandom strings.

**Protecting the intermediate results.** The function privacy also includes the protection of the intermediate results.

Intermediate results indicate the outputs from computing the previous circuits and being inputs of the following circuits. An intermediate result consists of garbled keys and thus the corresponding bits. As the intermediate results contains the internal runtime information, a malicious verifier may try to know it and gain the information about the content of circuits. But the intermediates result should also be hidden from the developer, who may be able to tamper the intermediate results and undermines the verification.

The problem about how to securely evaluate the intermediate results is a challenge for garbled circuits. The challenge originates from a fact: If the developer garbles each circuit by calling *GB.Garble* independently, the garbled keys for each circuit are irrelative. We can look back on how the gates in a circuit connect with each other. As mentioned in Section 2.1.2, sharing the same garbled keys between two immediate garbled truth tables realizes the computation between two adjacent gates. However, the garbled keys for two circuits are generated independently. Before being an input to the target circuit, the result of a circuit has to be decrypted, and then encoded by the garbled keys of target circuit. For example, a result $y'$ from a circuit $CT_i$ is the input to a circuit $CT_{i+1}$ (all circuits are indexed by topological order). $y'_i$ is encoded by $CT_i$'s garbled keys, so $y'_i$ has to be decrypted by the $CT_i$'s garbled keys first, and then is transformed into $x'_{i+1}$ by encoding the decrypted value $y_i$, with $CT_{i+1}$'s garbled keys.

The problem is that these operations can leak secrets to the verifier or the developer inevitably. The decryption action needs the mapping tuples between garbled keys and real values. If the verifier requests the developer to decrypt an intermediate result $y'_i$, the developer must learn the decrypted value $y_i$ once he decrypts $y'_i$. On the other hand, the verifier cannot acquire the mapping tuples to decrypt $y'_i$, or he would not just be able to know $y_i$, but use the mapping tuples to know the content of the wires and garbled truth tables.

In this case, we make use of two-input garbled circuits to solve the problem. The garbled circuits mentioned above are suited to our case: the inputs are only from the verifier. But in fact, the protocol of garbled circuits is often applied to deal with the two-input case, which provides the input privacy for the inputs from two parties. This feature can be used in our case. The mapping tuples is the input from the developer and the intermediate result is the input from the verifier. Garbled circuit for two-input case can provide the input privacy to both the verifier and the developer. On the other hand, the process of translating the intermediate result can also be performed by the garbled circuits. Through the circuit privacy of garbled circuits, the internal runtime information is under protection and the secrets is avoided leaking. The details will be specified in Section 3.5.1.

**Malicious behavior** A defect of Yao garbled circuits is that the security only holds on semi-honest cases and cannot carry over against malicious adversaries. But our protocol has to guarantee the protocol to be secure in malicious case, or the practical value of our work would be at a discount. Before designing a more secure protocol, we need to specify what kinds of behavior is malicious.

First of all, we should admit that it is impossible to enumerate all threats in real life. The attacks may come from arbitrary fields and unexpected – which humans cannot anticipate. Accordingly, it is meaningless to define specific threats rather than provide reliable mathematical definitions (Katz and Lindell, 2007, Chapter 1.4). Derived from the work (Cai *et al.*, 2016), we provide the explicit security definition against malicious behavior (Section 3.3.2). Definition 16 contains the restraints for both the verifier and the developer:

- The developer cannot do any changes to deviate the results from the correct procedures of verification, regardless of the intermediate results or external results;
- In the whole process of verification, the verifier cannot acquire any other information beyonds the external inputs, external outputs and the structure of the circuit graph, and cannot tamper the verification results.

Aimed at defending the malicious behavior of the developer, the protocol employs the cut-and-choose approach (Section 2.6). The usage lies in two aspects. The first aspect is that, the commitment scheme can help the verifier detect the inconsistent behavior with high probability, if a malicious developer uses different the garbled keys that are agreed with the ones generated in $GB.Garbled$. The commitment scheme involves the function $VS.Checker$, which is introduced in Section 3.4.5. The second aspect is that the malicious behavior can be detected with high chance if a malicious developer tampers the encoded inputs and changes the values.

The protocol should also contain the mechanisms to guard against the malicious behavior conducted by the verifier. The most possible case is that the verifier sends corrupt queries when he requests the developer to encode the inputs. In this way, we design a specific algorithm to check the queries, which is performed by Algorithm $Type$.

Besides, the verifier may denies the correct results of verification deliberately. Aimed at such a threat, we allow a third-party trusted verifier to repeat the original verification. This replication needs the running records of the original verification, and we call the records as a *Certificate*. A certificate is a piece of public information, and any third party can repeat previous verification by the corresponding certificate. Therefore, if a third-party trusted verifier runs the verification with the certificate, the second verification must be able to to detect the inconsistent results, and the malicious behavior like tampering the result of the original verification, must be detected.

## 3.2 Comparison with the Protocol Based on FHE

The topic, secure and trusted partial white-box verification, originates from the work of (Cai *et al.*, 2016). It is worthwhile to compare the Cai's protocol, which is based on Fully homomorphic encryption, with the our protocol, which is based on garbled circuits.

First, the two protocols adopt different means to represent the program. The protocol based on $FHE$ adopts tables (Section 2.1.1) to represent the program and thus executes the protocol on the tables. Accordingly, the table graph in Cai's work is also different from the circuit graph in our work. Since the protocol based on $FHE$ requires to works on single row tables, the original tables with multiple rows should be transformed into the single-row tables.

Second, the protocol based on $FHE$ uses different procedures to realize the secure verification. As $FHE$ enables computation to execute on encrypted data, an intermediate result can participate the next computation without the complicated translation like what we do in our protocol. From this point of view, it is an obvious advantage in Cai's work.

Compared to the protocol based on $FHE$, our protocol has the advantage in practicability. The current work reveals that only a partial version of fully homomorphic encryption can be implemented. For example, the representative work about implementing $FHE$ (Gentry and Halevi, 2011) only realized the "somewhat homomorphic encryption", which means a limited number of homomorphic operations. This fact reveals the bottleneck of $FHE$ in implementation. On the other hand, the work on the implementation of garbled circuits has been mature. Apart from the framework of Fairplay (Malkhi *et al.*, 2004), the work of (Huang *et al.*, 2011) and (Kreuter *et al.*, 2012) also presented their own frameworks to implement the garbled circuits. Other research, like the optimization of garbled circuits (Nielsen and Orlandi, 2009), has made attempts to apply garbled circuits to the large-scale industrial practice.

## 3.3 Scheme

### 3.3.1 Notations

For ease of accurate representation, our work involves many notations. Table 3.1 summarizes the important notations which appear in previous sections and will be used in following sections.

| Notations | Meanings |
|---|---|
| $G_c, G'_c$ | $G_c$ is a circuit graph consisting of a set of $n$ circuits $\{CT_i\}_{i=1}^n$ and a structure graph $G_{struc}$. $G'_c$ is the encrypted version of $G_c$. |
| $CT_i, CT'_{i,r}$ | $CT'_{i,r}$ is a garbled circuit for $CT_i$, where $i \in [1,n]$ and $j \in [1,s]$. In the cut-and-choose strategy, each $CT_i$ corresponds to $s$ garbled circuits, so $CT_{i,r}$ denotes $CT_i$'s $r_{th}$ garbled copies. |
| $IC, OC$ | $IC$ is a set of circuits accepting external inputs, while $OC$ is a set of circuits outputting external outputs. |
| $EX$ | $x \in EX$ is an external input to $G_c$. $EX$ is the complete set of external inputs provided by the verifier. External inputs are just the test cases generated from $VGA()$. |
| $w_{j,i}$ | $w_{j,i}$ denotes a wire in circuit $CT_i$, where $j \in [1, m_i]$. $m_i$ denotes the total wires in circuit $CT_i$. |
| $k^0_{i,j,r}$ or $k^1_{i,j,r}$ | In a garbled circuit $CT'_{i,r}$, if the bit value of a wire $w_{j,i}$ is 0, its garbled key is $k^0_{i,j,r}$; if the bit of $w_{j,i}$ is 1, $k^1_{i,j,r}$ is its garbled key.$i, j, r$ indicate the indices of circuits, wires and garbled copies respectively, where $i \in [1,n]$, $j \in [1, m_i]$ and $r \in [1,s]$. |
| $gk_{i,r}$ | $gk_{i,r}$ indicates a set of garbled keys for all input and output wires in $CT_{i,r}$, $gk_i = \{k^0_{j,i,r}, k^1_{j,i,r}\}_{j=1}^{l_i}$. |

| | |
|---|---|
| $x_i, x'_{i,r}$ | $x_i = (x_1, ..., x_{l_i})$ is the input to circuit $CT_i$, where $l_i$ indicates the bit length of $x_i$. $x'_{i,r} = (k_r^{x_{1,i}}, k_r^{x_{2,i}}, ..., k_r^{x_{l_i,i}})$ is the encrypted value of $x_i$ and the input to the garbled circuit $CT'_{i,r}$. |
| $y_i, y'_i$ | $y_i = (y_1, ..., y_{l_i})$ is the output from computing circuit $CT_i$ with $x_i$, with the same length $l_i$ as $x_i$. $y'_{i,r} = (k_r^{y_{1,i}}, k_r^{y_{2,i}}, ..., k_r^{y_{l_i,i}})$ is the encrypted value of $y_i$ and the output from $CT'_{i,r}$. |
| $map_{i,r}$ | $map_{i,r} = [(k^0_{i,j,r}, 0), (k^1_{i,j,r}, 1)]_{j=1}^{l_i}$ is a set of mapping tuples which contain the corresponding relationships between the garbled keys, only used for encoding inputs and outputs, and their bit values. |
| $Com(.)$ | $Com(.)$ is a function that takes a parameter as the input and generates the corresponding commitments. Particularly, $Com(input_i)$ denotes the set of commitments for $\{CT'_{i,r}\}_{r=1}^{s}$'s input wires $\{k^0_{j,i,r}, k^1_{j,i,r}\}_{j=1,r=1}^{l_i,s}$. |
| $Gr$ | $Gr$ is a pseudorandom generator |
| $sd_i$ | $sd_i$ is the seed for pseudorandom generator $Gr$, where $i$ is the circuit index. |
| $M$ | Memory $M$ can store and retrieve the record $(w, z, q, t)$ generated when encoding the inputs. A tuple $(w, z, q, t)$ indicates that a result $z$ is the circuit of computing $CT_q$ with an input $w$, and $z$ is the input to the next circuit $CT_t$. |
| $\tau_i$ | $\tau_i = (t_{1,i}, ..., t_{r,i}, ..., t_{s,i})$ is the challenge string during evaluating $CT_i$. If $t_{r,i} = 1$, the commitments of $CT'_{i,r}$ would be chosen to conduct consistency check. |

| | |
|---|---|
| $dc_{i,r}$ | $dc_{i,r}$ is a set of decommitments to decommit the commitments $Com(input_i)$ for garbled circuits $\{CT_{i,r}\}_{i=1,r=1}^{n,s}$. $dc_{i,r} = (\{\bar{k}_{i,j,r}^0, \bar{k}_{i,j,r}^1\}_{j=1}^{l_i}, sd_i)$, where $r$ is decided by $\tau_i$. $\{\bar{k}_{i,j,r}^0, \bar{k}_{i,j,r}^1\}_{j=1}^{l_i}$ are the garbled keys sent from $VS.Checker$. $sd_i$ is the seed for pseudorandom generator of generating commitments in circuit $CT_i$. |
| $\mathbb{R}_i$ | If $t_{r,i} \in \tau_i$ and $t_{r,i} = 1$, the index $r \in \mathbb{R}_i$. |
| $QA_E$ | $QA_E = (Q_E, A_E)$ are the interaction records during executing $VS.Encode$. $Q_E = (w, z, s, t)$ are records of queries sent from the verifier. $A_E$ is the response of $VS.Encode$ from the developer. |
| $QA_C$ | $QA_C = (Q_C, A_C)$ are the interaction records during executing $VS.Checker$. $Q_C = (\tau_i)$ are the queries sent from the verifier. $A_C$ is the response of $VS.Checker$ from the developer, $A_C$ is exactly the records of $dc_{i,r}$. |

Table 3.1: Notations

### 3.3.2 Definitions for the Scheme

**Definition 14** (Scheme for partial white-box verification)**.** *A scheme for secure and trusted partial white-box verification is a tuple of p.p.t algorithms such that*

***VS.Encrypt(**$G_c, 1^K$**):*** *VS.Encrypt is an algorithm that takes as input a set of circuits $\{CT_1, ..., CT_n\}$ and a security parameter $1^K$, and outputting a set of garbled circuits $\{CT'_{i,r}\}_{i=1,r=1}^{n,s}$, a complete set of garbled keys $\{k_{i,j,r}^0, k_{i,j,r}^1\}_{i=1,j=1,r=1}^{n,m_i,s}$ for all wires in each circuit, and the commitments for all input wires in each circuit $\{Com(input_i)\}_{i=1}^n = \{Com(gk_{i,r})\}_{i=1,r=1}^{n,s}$ where $gk_{i,r} = \{k_{i,j,r}^0, k_{i,j,r}^1\}_{j=1}^{l_i}$.*

***VS.Encode(*$gk_i, x_i$*):*** *VS.Encode is an algorithm that takes as input the set of garbled keys* $gk_{i,r} = \{k^0_{j,i,r}, k^1_{j,i,r}\}^{l_i}_{j=1}$ *and* $x_i$*, outputting a garbled input,* $x'_{i,r}$*.*

***VS.Checker(*$\tau_i$*):*** *VS.Checker is an algorithm takes as input a challenge string* $\tau_i = (t_{1,i}, ..., t_{s,i})$*, and returns a set of decommitments* $\{dc_{i,r}\}_{r \in \mathbb{R}_i}$*, where* $\mathbb{R}_i$ *is a set of* $r$*'s which satisfy* $t_{r,i} \in \tau_i, t_{r,i} = 1$*.*

***VS.Eval(*$1^K, QA_E, QA_C, G'_c$*):*** *VS.Eval is an algorithm that take as input a security parameter* $1^K$*, records of VS.Encode* $QA_E$*, records of VS.Checker* $QA_C$ *and the encrypted version of objective circuit graph* $G'_c$*, returning 0 or 1 to indicate whether the verification succeeds or not.*

Having defined the strategies and security requirements of evaluating the protocol in Section 2.2, we can provide the definitions for correctness and security.

**Definition 15** (Correctness). *A verification scheme is* correct *iff a trusted third-party verifier* $V'$ *and a real-life verifier* $V$ *can obtain such results:* $VS.Eval(1^K, QA_E, QA_C, G'_c) = 1$ *if and only if both of the following equations hold:*

$$Pr_r[V(1^K, G_c, G_{spec}, VGA, CP)(r) = V(1^K, G'_c, G_{spec}, VGA, CP)(r)] \geq 1 - negl(K) \tag{3.1}$$

$$Pr_r[V(1^K, G'_c, G_{spec}, VGA, CP)(r) = V'(1^K, G'_c, G_{spec}, VGA, CP)(r)] \geq 1 - negl(K) \tag{3.2}$$

*where* $V$ *is the verifier hardwired in* $VS.Eval$*,* $r$ *a string of random bits and* $G_c$ *is the original circuit graph.*

**Definition 16** (Security). *For* $A = (A_1, A_2)$ *and* $S = (S_1, S_2, S_3)$ *which are tuples of p.p.t algorithms, consider the two experiments in Table 3.2,3.3.*

| $Exp^{real}(1^K)$ |
|---|
| 1. $(G_c, CP, state_A) \leftarrow A_1(1^K)$ |
| 2. $G'_c \leftarrow VS.Encrypt(1^K, G_c)$ |
| 3. $\alpha \leftarrow A_2^{VS.Encode, VS.Checker}(1^K, G'_c, G_c, CP, state_A)$ |
| 4. Output $\alpha$ |

Table 3.2: $Exp_1$: real experiment

| $Exp^{ideal}(1^K)$ |
|---|
| 1. $(G_c, CP, state_A) \leftarrow A_1(1^K)$ |
| 2. $\tilde{G}_c \leftarrow S_1(1^K)$ |
| 3. $\alpha \leftarrow A_2^{S_2^{O_1}, S_3^{O_2}}(1^K, \tilde{G}_c, G_c, CP, state_A)$ |
| 4. Output $\alpha$ |

Table 3.3: $Exp_2$: Ideal experiment

*Simulators $S_1, S_2, S_3$ are used to simulate $VS.Encrypt, VS.Encode$ and $VS.Checker$. $S_2, S_3$ are augmented with access to oracle $O_1, O_2$:*

1. *$O_1$ provides the information as $VS.Encode$ obtains in $Exp_{real}$, except the original value $x_i$ of the intermediate input $\{x'_{i,r}\}_{r=1}^{s}$.*
2. *$O_2$ provides the information as $VS.Checker$ obtains in $Exp_{real}$, except the decommitments $\{dc_{i,r}\}_{r \in \mathbb{R}_i}$, where $\mathbb{R}_i$ is a set of $r$'s which satisfy $t_{r,i} \in \tau_i, t_{r,i} = 1$.*

*A verification scheme $VS$ is* secure *if there exists a tuple of p.p.t. simulators $S = (S_1, S_2, S_3)$ and oracles $O_1, O_2$ such that for all pairs of p.p.t. adversaries $A = (A_1, A_2)$, the following is true for any p.p.t. algorithm $D$:*

$$\left|Pr[D(Exp^{ideal}(1^K), 1^K) = 1] - Pr[D(Exp^{real}(1^K), 1^K) = 1]\right| \leq negl(K),$$

*i.e. the two experiments are computationally indistinguishable.*

*Note: Memory state $State_A$ is the information of communication between adversaries $\{A_1, A_2\}$. $\alpha$ represents the view outputted from the experiments.*

## 3.4 Designing the Protocol

Now we can design the the protocol for partial white-box verification. The ideas are derived from the analysis in Section 3.1.

### 3.4.1 Transforming the Program into Circuits

At the very beginning, we need to make the program represented by the desired means of description, Boolean circuits (Snyder, 2014). It is a necessary step to apply the garbled circuits to our construction. According to the Assumption 3, it is plausible to transform a program described by tables into the corresponding circuits. Through this step, the original table graph $G_t$ becomes a circuit graph, $G_c$ (Section 2.1). We skip specifying how to implement the transformation, as it is function specific and beyond our discussion scope.

### 3.4.2 Encrypting the Circuits

The first step of the protocol is to encrypt the circuits. The developer calls $VS.Encrypt$ to encrypt the circuit graph $G_c$. In fact, $VS.Encrypt$ applies $GB.Garble$ to garbling each circuit $CT_i$. By the cut-and-choose strategy, the developer generates $s$ different garbled circuits for a single $CT_i$. Calling $GB.Garble$ also generates the garbled keys for all wires $\{w_{i,j}\}_{i=1,j=1}^{n,m_i}$ of each circuit, where $m_i$ denotes the total number of wires in $CT_i$.

Another job of $VS.Encrypt$ is to generate commitments for garbled keys. We are curious about a certain part of garbled keys, $\{k_{i,j,r}^0, k_{i,j,r}^1\}_{i=1,j=1,r=1}^{n,l_i,s}$. These garbled keys are used to encode the inputs and outputs. $VS.Encrypt$ commits these garbled keys and obtains the commitments $\{com(gk_{i,r})\}_{i=1,r=1}^{n,s} = \{com(k_{i,j,r}^0), com(k_{i,j,r}^1)\}_{i=1,j=1,r=1}^{l_i,n,s}$.

### 3.4.3 Encoding

$VS.Encode$ has two jobs. The first one is to encode an input with its corresponding garbled keys. The fact of encoding is to replace each bit $x_{j,i}$ of input $x_i = (x_{1,i}, ..., x_{l_i,i})$ by the garbled keys $k_{i,r}^{x_{j,i}}$. Given an external input $x_i$, the verifier can directly call enter $GB.Encode$, obtaining the encoded input $x'_{i,r} = (k_r^{x_{1,i}}, k_r^{x_{2,i}}, ..., k_r^{x_{l_i,i}})$. If the input is an intermediate result $y'_{i,r}$ from garbled circuit $CT'_{i,r}$, the verifier calls $TF()$ (Algorithm 2) to translate the intermediate input into the correct encoded input.

The second job is to prevent the malicious queries from the verifier, as mentioned in Section 3.1. Thus, before calling $GB.Encode$ or $TF$ to encode an input, the verifier invokes $Type()$ to check if the input is out of place. In the meantime, $Type$ also involves the consistency check. Through retrieving the tuple $(w, z, q, t)$ with the same circuit indices $q$ and $t$ from memory $M$, $Type$ can detect if the current input is consistent with the previous record. $Type$ is introduced in Section 3.5.2.

### 3.4.4 Evaluation

Section 2.4.2 has stated the mechanisms of how to evaluate a garbled circuit with an input. The process of evaluation within a circuit is actually running on the garbled truth tables, as shown in Figure 2.6. Calling $GB.Eval$ to evaluate an individual circuit is the same effect. Evaluating across circuits involves the transformation of different garbled keys to encode the intermediate values, which is exactly realized by $VS.Encode$. Then the next round of evaluating within the circuit starts again.

When the process continues until the end circuits, the verification should generate a decrypted result rather than cipher. Section 2.4.2 specifies two ways to decrypt a garbled output. In our situation, we follow the trend that the end gates can output real bits directly. Therefore, our protocol save an extra step to decrypt the garbled outputs.

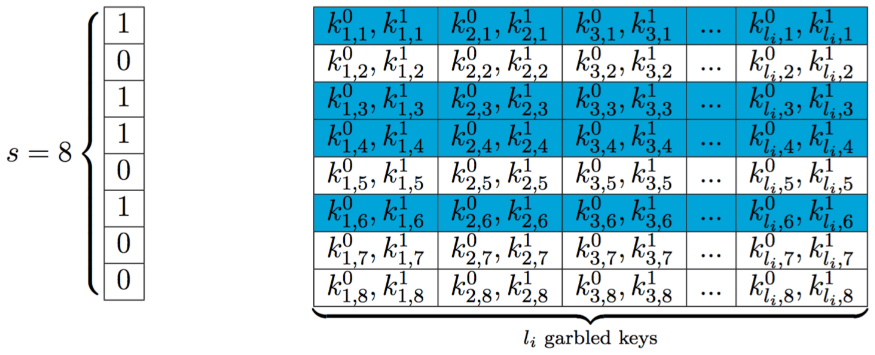
Considering the cut-and-choose strategy, the $s$ garbled circuits for $CT_i$ will generates $s$ garbled outputs. After evaluating $\{CT_{i,r}\}_{r=1}^{s}$ for each round, the verifier can check whether the results $\{y_{i,r}\}_{r=1}^{s}$ are consistent. If the check fails, the verifier can abort the verification. Among the garbled results like $\{y_{i,r}\}_{r=1}^{s}$, the verifier can choose any of them to be the input to next circuit.

### 3.4.5 Checking the Commitments

As described in Section 2.7, we apply the *bit-commitment scheme* to check the data integrity. The bit-commitment scheme has two phases, the "commit stage" and the "reveal stage". In our protocol, the "commit stage" is the step of calling $VS.Encrypt$. The "reveal stage" is the step of calling $VS.Checker$ and the following consistency checks.

Before the "reveal stage", the two parties should take rounds of interaction to set a *challenge string* $\tau$. This challenge string is to indicate which circuits are used for being checked and which circuits are used for being evaluated. In order to prevent the two parties affecting fairness of deciding the challenge string, the protocol employs the *coin-tossing protocol* to generate it. As we adopt the cut-and-choose strategy, a circuit $CT_i$ has $s$ garbled copies, $\{CT_{i,1}, ..., CT_{i,s}\}$. In a challenge string $\tau_i = (t_{1,i}, ..., t_{r,i}, ..., t_{s,i})$, if $t_{r,i} = 1$, the commitments and garbled keys for $CT'_{i,r}$ will be chosen to check; if $t_{r,i} = 0$, the garbled keys for $CT'_{i,r}$ will be chosen to evaluate

$CT'_{i,r}$, which is shown in the steps of "DECOMMITTING,CHECKING" and "EVALUATION" in Algorithm 4. An example about the challenge string is illustrated in Figure 3.1.


$s = 8$
1
0
1
1
0
1
0
0
$k^0_{1,1}, k^1_{1,1}$ | $k^0_{2,1}, k^1_{2,1}$ | $k^0_{3,1}, k^1_{3,1}$ | ... | $k^0_{l_i,1}, k^1_{l_i,1}$
$k^0_{1,2}, k^1_{1,2}$ | $k^0_{2,2}, k^1_{2,2}$ | $k^0_{3,2}, k^1_{3,2}$ | ... | $k^0_{l_i,2}, k^1_{l_i,2}$
$k^0_{1,3}, k^1_{1,3}$ | $k^0_{2,3}, k^1_{2,3}$ | $k^0_{3,3}, k^1_{3,3}$ | ... | $k^0_{l_i,3}, k^1_{l_i,3}$
$k^0_{1,4}, k^1_{1,4}$ | $k^0_{2,4}, k^1_{2,4}$ | $k^0_{3,4}, k^1_{3,4}$ | ... | $k^0_{l_i,4}, k^1_{l_i,4}$
$k^0_{1,5}, k^1_{1,5}$ | $k^0_{2,5}, k^1_{2,5}$ | $k^0_{3,5}, k^1_{3,5}$ | ... | $k^0_{l_i,5}, k^1_{l_i,5}$
$k^0_{1,6}, k^1_{1,6}$ | $k^0_{2,6}, k^1_{2,6}$ | $k^0_{3,6}, k^1_{3,6}$ | ... | $k^0_{l_i,6}, k^1_{l_i,6}$
$k^0_{1,7}, k^1_{1,7}$ | $k^0_{2,7}, k^1_{2,7}$ | $k^0_{3,7}, k^1_{3,7}$ | ... | $k^0_{l_i,7}, k^1_{l_i,7}$
$k^0_{1,8}, k^1_{1,8}$ | $k^0_{2,8}, k^1_{2,8}$ | $k^0_{3,8}, k^1_{3,8}$ | ... | $k^0_{l_i,8}, k^1_{l_i,8}$
$l_i$ garbled keys

Note: In this example, $\tau = (10110100)$, $s = 8$. $l_i$ denotes the length of $x_i$. The garbled keys in blue are chosen to be checked.

Figure 3.1: An example of the challenge string

We design *VS.Checker* to perform the "reveal stage" in the bit-commitment scheme. The *decommitments* is the "keys" to "decrypt" the commitments and reveal the committed values. What *VS.Checker* should do is to send the decommitments to the verifier. Which set of decommitments to be sent is decided by the challenge string $\tau$. The verifier can use the decommitments to conduct such checks: 1. Given the decommitments, the verifier can build a new set of commitments by the same method as generating the original commitments. Thus, he can check whether the new commitments are equal to the original commitments. 2. Through decommitting the commitments, the verifier obtains the original garbled keys. So the verifier can check if the garbled keys in $x'_{i,r}$ match with any of original garbled keys. If there exist garbled keys in $x'_{i,r}$ but not in the original garbled keys, it proves that the developer may fabricate the garbled keys. The idea is shown in Table 3.4 and 3.5.

Note: It is assumed that $x'_{i,r} = (k_r^{x_{1,i}}, k_r^{x_{2,i}}, ..., k_r^{x_{l_i,i}})$, and its original value is $x = (011, ..., 0)$. So the check is to see if the garbled keys in Table 3.4 exist in Table 3.5.

| Bit values | $w_{1,i}$ | $w_{2,i}$ | $w_{3,i}$ | ... | $w_{l_i,i}$ |
|---|---|---|---|---|---|
| 0 | $k_r^{x_{1,i}}$ | - | - | ... | $k_r^{x_{l_i,i}}$ |
| 1 | - | $k_r^{x_{2,i}}$ | $k_r^{x_{3,i}}$ | ... | - |

Table 3.4: The mapping relations between garbled keys and bits in $x'_{i,r}$

| Bit values | $w_{1,i}$ | $w_{2,i}$ | $w_{3,i}$ | ... | $w_{l_i,i}$ |
|---|---|---|---|---|---|
| 0 | $k^0_{i,1,r}$ | $k^0_{i,2,r}$ | $k^0_{i,3,r}$ | ... | $k^0_{i,l_i,r}$ |
| 1 | $k^1_{i,1,r}$ | $k^1_{i,2,r}$ | $k^1_{i,3,r}$ | ... | $k^1_{i,l_i,r}$ |

Table 3.5: The garbled keys from opened commitments

### 3.4.6 Third Party Verification

As mentioned in Section 3.1, the protocol provides a publicly known information, a *certificate*, to repeat the previous verification by a third-party trusted verifier. The certificate consists of two parts. One part is the records of all the original garbled circuits and commitments. The other part is the transcripts of the communication between the verifier and the developer. These transcripts are produced during executing $VS.Encode$ and $VS.Checker$. Thus, in the communication transcripts $QA_E$ of calling $VS.Encode$, we denote the queries from the verifier as $Q_E$, the response from the developer as $A_E$. Likewise, the communication transcripts of calling $VS.Checker$ can be denoted as $QA_C$. The queries from the verifier is represented as $Q_C$ and the response from the developer is represented as $A_C$.

$VS.Eval$ is designed to finish the third-party verification. The detailed implementation is specified in Algorithm 8. The ideas behind $VS.Eval$ is that repeating the original verification with the certificate can detect if any results are not consistent with the current execution. In this case, the new verifier should not be any of the parties executing the previous verification, and he should be trusted to be a authoritative verifier. Such a third-party trusted verifier is defined in Definition 7, which is denoted as $V'$.

Eventually, $V'$ should also evaluate the specification, which is stipulated in the strategy of verification (Section 2.2). Regarding the specification as a circuit graph $G_{spec}$, $V'$ can evaluate $G_{spec}$ with the same external inputs $EX$ and conduct the checks like, if the evaluation paths on $G_{spec}$ corresponds to the verification paths on $G'_c$, or if the final result of the verification is the same as the final result of evaluating $G_{spec}$.

## 3.5 Auxiliary Functions

### 3.5.1 TF

As introduced in Section 1.3 and Section 3.1, we need to design a function that realizes securely encoding intermediate results without leaking secrets of two parties to the opponent. It is assumed that the two sets of garbled circuits $\{CT'_{u,r}\}_{j=1}^{l_u}$ and $\{CT'_{v,r}\}_{j=1}^{l_v}$ participate the process. The inputs of the two parties are: the verifier provides an intermediate result $y'_{u,r}$; the developer provides the mapping tuples $map_{u,r}$ for the output wires of $\{CT'_{u,r}\}_{j=1}^{l_u}$, and another mapping tuples for the input wires of $\{CT'_{v,r}\}_{j=1}^{l_v}$.

The functionality of $TF$ is divided into two parts. The first part is to decrypt the intermediate result $y'_{u,r}$. With the mapping tuples $map_{u,r} = [(k^0_{u,j,r}, 0), (k^1_{u,j,r}, 1)]_{j=1}^{l_u}$ from the developer, the real value behind $y'_{u,r}$ can be computed and denoted as $y_u$.

The other part of $TF$ is to encode $y_u$ with the garbled keys of $\{CT_{v,r}\}_{r=1}^{s}$. With the mapping tuples $map_{v,r} = [(k_{v,j,r}^{0}, 0), (k_{v,j,r}^{1}, 1)]_{j=1}^{l_v}$, each bit $y_{j,u} \in y_u$ can be replaced by corresponding garbled keys. Combined the steps into one, the process of $TF$ can be perceived as translating an encrypted value, which is first encoded by a set of garbled keys, into a different value, which is also encoded by another set of garbled keys, but the bit values behind the two encrypted values are the same. Table 3.6 depicts the mapping relations.

| real values | garbled values of $CT_u$ | garbled values of $CT_v$ |
|---|---|---|
| 0 | $k_{j,u}^{0}$ | $k_{j,v}^{0}$ |
| 1 | $k_{j,u}^{1}$ | $k_{j,v}^{1}$ |

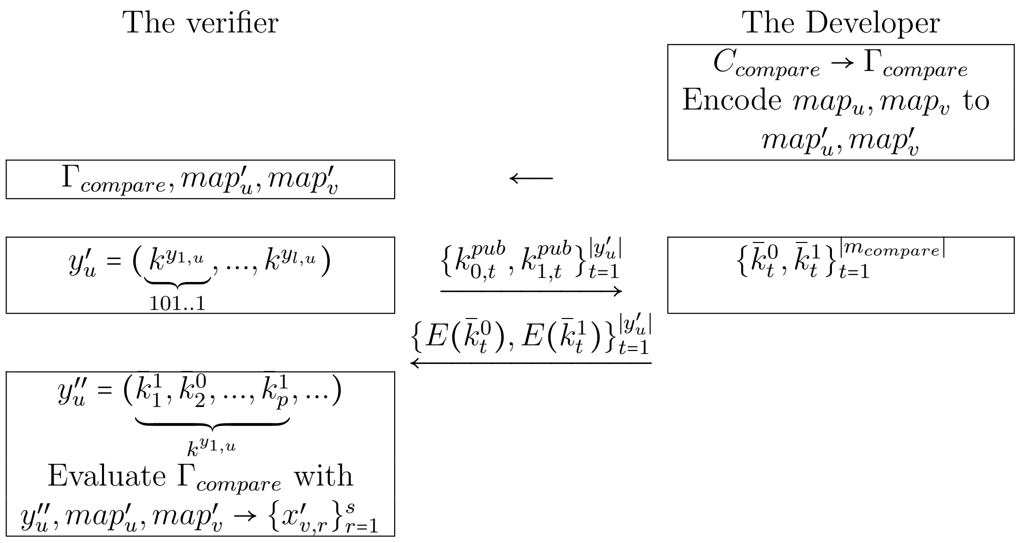
Table 3.6: Mapping tuples between $CT_u$ and $CT_v$

Thus, we have such security requirements:

> The intermediate result $y'_{u,r}$ and the mapping tuples $\{map_{u,r}\}_{r=1}^{s}$ and $\{map_{v,r}\}_{r=1}^{s}$ are sensitive information and owned privately by the verifier and the developer respectively. In the meantime, the above execution of decrypting $y'_{u,r}$ and encoding $y_v$ should be confidential to both the verifier and the developer.

To securely realize $TF$, we introduce a two-party garbled circuit. First, the process of decrypting $y'_{u,r}$ and encoding $y_v$ can be wrapped as a function *compare*. This function is doing the substantive job of $TF$. Then, the function *compare* is garbled by the developer, and the garbled keys $gk_{compare}$ for garbled circuit $\Gamma_{compare}$ are generated. The developer uses the garbled keys $gk_{compare}$ to encode its inputs, the mapping tuples $\{map_{u,r}\}_{r=1}^{s}$ and $\{map_{v,r}\}_{r=1}^{s}$, and sends the encoded inputs to the verifier as well as the garbled circuit $\Gamma_{compare}$. Subsequently, the verifier encodes his input $y'_{u,r}$ via oblivious transfer ($OT$, see Section 2.5). As specified in **Step 2** in Section 2.4.2, the verifier exchanges each bit in $y'_{u,r}$ for the corresponding garbled key from $gk_{compare}$. In this way, the verifier obtains the encoded input $y''$. Finally, the verifier uses his encoded input $y''_{u,r}$ to compute the garbled circuit $\Gamma_{compare}$ as well as the garbled inputs from the developer. The results of this computation would be an encoded input $\{x'_{v,r}\}_{r=1}^{s}$ to the circuit $\{CT_{v,r}\}_{r=1}^{s}$.

In terms of the security, the inputs privacy of the two parties is protected by the garbled keys, while the function security of *compare* is guaranteed by garbled circuit $\Gamma_{compare}$. $TF$ is realized in Algorithm 2. Similar to Figure 2.9, the principles of $TF$ can be depicted as an interaction diagram (Figure 3.2).


The verifier
The Developer
$C_{compare} \rightarrow \Gamma_{compare}$
Encode $map_u, map_v$ to $map'_u, map'_v$
$\Gamma_{compare}, map'_u, map'_v$
$y'_u = (\underbrace{k^{y_{1,u}}}_{101..1}, ..., k^{y_{l,u}})$
$\{k^{pub}_{0,t}, k^{pub}_{1,t}\}_{t=1}^{|y'_u|}$
$\{\bar{k}^0_t, \bar{k}^1_t\}_{t=1}^{|m_{compare}|}$
$\{E(\bar{k}^0_t), E(\bar{k}^1_t)\}_{t=1}^{|y'_u|}$
$y''_u = (\underbrace{k^1_1, k^0_2, ..., k^1_p}_{k^{y_{1,u}}}, ...)$
Evaluate $\Gamma_{compare}$ with $y''_u, map'_u, map'_v \rightarrow \{x'_{v,r}\}_{r=1}^{s}$

This diagram includes an example. It is assumed that garbled key $k^{y_{1,u}} = 101..1$. After the job of $OT$, $k^{y_{1,u}} = 101..1$ are encoded into $\bar{k}^1_1, \bar{k}^0_2, ..., \bar{k}^1_p$.

Figure 3.2: The interaction diagram of function $TF$

**Algorithm 2** $TF(y'_{u,r}, \{map_{u,r}\}_{r=1}^{s}, \{map_{v,r}\}_{r=1}^{s}, u, v)$. $y'_{u,r}$ comes from the verifier. $\{map_{u,r}\}_{r=1}^{s}$ and $\{map_{v,r}\}_{r=1}^{s}$ come from the developer. $u, v$ are circuit indices and publicly known.

```
1: Garble compare: $GB.Garble(1^K, C_{compare}) = \Gamma_{compare}, gk_{compare}$   ▷ $C_{compare}$ is a Boolean circuit with the functionality of function compare.
2: $y'_{u,r}$ is encoded into $y''_{u,r}$ by garbled keys $gk_{compare} = \{\bar{k}_t^0, \bar{k}_t^1\}_{t=1}^{m_{compare}}$   ▷ $m_{compare}$ indicates the total number of wires in $C_{compare}$. $t$ is the index for garbled keys of $\Gamma_{compare}$.
3: Evaluate $\Gamma_{compare}$: $GB.Eval(\Gamma_{compare}, y''_{u,r}, \{map_{u,r}\}_{r=1}^{s}, \{map_{v,r}\}_{r=1}^{s}, u, v)$. That is, compute $compare(y'_{u,r}, \{map_{u,r}\}_{r=1}^{s}, \{map_{v,r}\}_{r=1}^{s}, u, v)$
4: return $\{x'_{v,r}\}_{r=1}^{s}$.
5:
6: Build a function, $compare(y'_r, \{map_{u,r}\}_{r=1}^{s}, \{map_{v,r}\}_{r=1}^{s}, u, v)$   ▷ $y'_{u,r} = (k_r^{y_{1,u}}, k_r^{y_{2,u}}, ..., k_r^{y_{l_u,u}})$. Assumed that $k_r^{y_{j,u}} \in y'_{u,r}$'s length is $p$, the length of $y'_{u,r}$ would be $l_u \cdot p$.
7: function COMPARE($y'_{u,r}, \{map_{u,r}\}_{r=1}^{s}, \{map_{v,r}\}_{r=1}^{s}, u, v$)
8:     Replace $k_r^{y_{j,u}}$ by its corresponding bit $y_{j,u}$ from $\{map_{u,r}\}_{r=1}^{s}$.
9:     Obtain the bit sequence of $y_u = (y_{1,u}, ..., y_{j,u}, ...y_{l_u,u})$, where $y_{j,u} \in [0,1]$
10:    $x_v = y_u = (x_{1,v}, ..., x_{j,v}, ...x_{l_v,v})$, where $x_{j,v} = y_{j,u}$
11:    for $r = 1$ to $s$ do
12:        Replace $x_{j,v}$ by the corresponding garbled key $k_r^{y_{j,v}}$ from $\{map_{v,r}\}_{r=1}^{s}$
13:        Obtain an encoded input $x'_{v,r} = (k_r^{x_{1,v}}, k_r^{x_{2,v}}, ..., k_r^{x_{l_v,v}})$
14:    end for
15:    return $\{x'_{v,r}\}_{r=1}^{s}$
16: end function
```

### 3.5.2 Type

*Type* is used to judge the types of inputs sent from the verifier. Similar to $TF$, *Type* takes as input the indices of circuits, $u, v$ and an input $x$. The meaning behind $u, v$ is that an input $x_v$ of the circuit $CT_v$ is the output of the circuit $CT_u$.

Based on the topological structure of $G_{struc}$, *Type* can check if the received input $x$ corresponds to the locations which $u, v$ indicate:

- If $u, v$ indicate that $x$ is an external input, $x$ should be an unencrypted value.
- If $u, v$ indicate that $x$ is an intermediate input, $x$ should be an encoded value and consist of garbled keys.

To conduct more advanced check, like consistency check, *Type* can recall the previous records to check if the input $x$ exists ever in previous computation. It is assumed that the protocol has a *memory* $M$ which stores previous computational record, in a form of $(w, z, q, t)$. In such a tuple $(w, z, q, t)$, $w$ is the garbled input to garbled circuits $\{CT'_{q,r}\}_{r=1}^{s}$, while $z$ is the output of computing $\{CT'_{q,r}\}_{r=1}^{s}$ and will be the input to $\{CT'_{T,r}\}_{r=1}^{s}$. These records are produced when evaluating garbled circuits with the encoded inputs, as specified in Section 3.4.4. *Type* can make use of $M, G_{struc}$ to conduct the following checks:

- Given $u, v$, *Type* can take out all the outputs which are incident from $\{CT_{u,r}\}_{r=1}^{s}$ in $M$, and see if $x$ is equal to any of these outputs. If $x$ does not correspond to any of outputs from $\{CT_{u,r}\}_{r=1}^{s}$, the check detects that the intermediate input $x$ may be fabricated by the verifier.

  For example, $(x'_{u,r}, y'_{u,r}, q, u)$ is a tuple taken from memory $M$. When the verifier sends the encoded input $y''_{u,r}$ (suppose that its next circuit is $\{CT_{v,r}\}_{r=1}^{s}$) and claim that $y''_{u,r}$ is the output of computing $\{CT_{u,r}\}_{r=1}^{s}$. $Type$ can check if $y'_{u,r}$ is equal to $y''_{u,r}$ when $q = u$.

- Given $u, v$, $Type$ can take out all the circuits, like $\{CT_{i,r}\}_{i=1,r=1}^{n',s}$, that the current verification passes by, where $n'$ is the biggest circuit index in topological order and $n' < n$. If the previous data from the memory $M$ did not reach the circuits $u, v$, e.g. $n' < u$ or $n' < v$, the process of verification must be intermittent and $x$ may be fabricated by the verifier.

**Algorithm 3** $Type(u, v, x)$ is to recognize the type of input $x$ and check if $x$ is valid. Returning 0 means $x$ is not a valid input; returning 1 means $x$ is a valid external input; returning 2 means $x$ is a valid intermediate input.
A tuple $(w, z, q, t)$ is the records retrieved from memory $M$. The tuple means that a computational result $z$ is outputted from computing circuit $CT_q$ with its input $w$, and $z$ is the intermediate input to the next circuit $CT_t$.

```
1: if In each tuples $(w, z, q, t)$ from $M$, there are no circuit index $t$ equal to $u$ then
   ▷ $u$ has not been traversed
2:     return 0
3: end if
4: if $u, v$ indicate $x$ is external input but $x$ is an encrypted value then
5:     return 0
6: else
7:     return 1
8: end if
9: if $u, v$ indicate $x$ is an intermediate input and $x$ consists of garbled keys then
10:     for each tuples $(w, z, q, t)$ from $M$ do
11:         if there exists no circuit index $t = u$ and previous outputs $z = x$ then
12:             return 0
13:         end if
14:     end for
15: else
16:     return 2
17: end if
```

## 3.6 Construction for Malicious Cases

Based on the work in previous sections, we present the construction for the protocol of secure and trusted partial white-box verification. The construction are described by pseudocode.

In next section, Algorithms 4 to 7 implement the designs in Section 3.4. Algorithm 4 specifies the main procedure of the protocol. Algorithm 5, 6, 7 and 8 implement the key steps in Algorithm 4. To illustrate the procedures better, Section 3.6.2 provides an interaction diagram to specify how the two parties interact in the protocol. Some steps in the interaction diagram are numbered in circle, like ①②. These notations are used to mark the steps which can correspond to certain pieces of algorithms of next section.

### 3.6.1 Algorithmic representation for protocol

**Algorithm 4** Main Procedures of the Protocol

```
1: Setting:
2: 1.Transform the program from a table graph $G_t$ into a circuit graph $G_c$.
3: 2.The execution follows the topological order $i$, so the verification starts from $CT_1$ and ends at $CT_n$.
4: 3.$EX$ represents the set of external inputs (or test cases).
5: ① ENCRYPTING:
6: $VS.Encrypt(1^K, G_c) \rightarrow$
7: $(G'_c, \{k^0_{i,j,r}, k^1_{i,j,r}\}^{n,m_i,s}_{i=1,j=1,r=1}, \{CT'_{i,r}\}^{n,s}_{i=1,r=1}, \{Com(input_i)\}^n_{i=1})$   ▷ Where $i$ denotes the index of circuits in the circuit graph $G_c$, $j$ denotes the index of wires in circuit $CT_i$, and $r$ denotes the index of circuits in the $s$ garbled copies of $CT_i$.
8: $\{k^0_{i,j,r}, k^1_{i,j,r}\}^{n,m_i,s}_{i=1,j=1,r=1}$ denote the complete sets of garbled keys for all the wires in every circuit, where $m_i$ indicates the total number of wires in $CT_i$.
9: $Com(input_i)$ denotes the commitments for all input wires in $\{CT'_{i,r}\}^s_{r=1}$, i.e. $Com(input_i) = \{k^0_{j,i,r}, k^1_{j,i,r}\}^{l_i,s}_{j=1,r=1}$.
10: Send $G'_c, \{Com(input_i)\}^n_{i=1}$ to the verifier
11:
12: ② ENCODING:
13: while traversing circuits $CT_i$ in $G_{struc}$ do
14:     Take $x$ from $EX$ OR an intermediate result $y'_{i,r}$
15:     $\{x'_{i,r}\}^s_{r=1} \leftarrow VS.Encode(x)$ OR $\{x'_{i,r}\}^s_{r=1} \leftarrow VS.Encode(y'_{i,r})$
16:     The verifier obtains the encoded inputs $\{x'_{i,r}\}^s_{r=1}$   ▷ An encoded input $x'_{i,r}$ is illustrated as Table 3.4
```

```
17: ③ **SETTING A CHALLENGE STRING**:    ▷ Execute the coin-tossing protocol
18: **for** $r$ from 1 to $s$ **do**
19:     The developer chooses a random bit $\sigma_{r,i} \in \{0,1\}$ and a seed $seed_{r,i} \in \{0,1\}^n$, where $seed_{r,i}$ is used in generating commitment for $\sigma_{r,i}$. Then the developer sends the commitment of $\sigma_{r,i}$, $com(\sigma_{r,i})$ to the verifier.
20:     The verifier chooses a random bit $\sigma'_{r,i} \in \{0,1\}$ and sends it to the developer.
21:     The developer computes a bit value $\tau_{r,i} = \sigma_{r,i} \oplus \sigma'_{r,i}$ and sends $\sigma_{r,i}$ and $seed_{r,i}$ to the verifier.
22:     The verifier receives $\sigma_{r,i}$ and $seed_{r,i}$ and verifies whether he can generate the same commitment, by $seed_{r,i}$ and $\sigma_{r,i}$, as $com(\sigma_{r,i})$.
23: **end for**
24: $\tau_i = (\tau_{i,1}, ...\tau_{r,i}, ...\tau_{s,i})$
25:
26: ④⑤ **DECOMMITTING, CHECKING**:
27: $(\{\bar{k}^0_{i,j,r}, \bar{k}^1_{i,j,r}\}^{l_i}_{j=1}, \overline{sd}_i)_{r\in\mathbb{R}} = \{dc_{i,r}\}_{r\in\mathbb{R}} \leftarrow VS.Checker(\tau_i)$, where $\mathbb{R}_i$ is a set of $r$'s which satisfy $t_{r,i} \in \tau_i, t_{r,i} = 1$.    ▷ $VS.Checker$ only returns the decommitments chosen in the challenge string $\tau_i$.
28: **for all** index $r$ where $t_{r,i} = 1$ in $\tau_i = (t_{1.i}, ..., t_{s,i})$ **do**
29:     The verifier computes a new commitment $Com(\overline{input}_i)$ by receiving $\{dc_{i,r}\}_{r\in\mathbb{R}_i} = \{\bar{k}^0_{i,j,r}, \bar{k}^1_{i,j,r}\}^{l_i}_{j=1,r\in\mathbb{R}_i}, \overline{sd}_i$.
30:     **if** $Com(\overline{input}_i)$ is not equal to $Com(input_i)$ **then**
31:         **return** 0
```

```
32:         end if
33:         The verifier decommits the $Com(input_i)$ by the decommitments $\{dc_{i,r}\}_{r\in\mathbb{R}_i}$,
        and obtains the sets of garbled keys for the inputs of $CT_i$, i.e. $\{k^0_{j,i,r}, k^1_{j,i,r}\}^{l_i}_{j=1,r\in\mathbb{R}_i}$.
34:         if the garbled keys in $\{x'_{i,r}\}_{r\in\mathbb{R}_i} = (k_r^{x'_{i,1}}, ..., k_r^{x'_{i,l_i}})_{r\in\mathbb{R}_i}$ do not appear in
        $\{k^0_{j,i,r}, k^1_{j,i,r}\}^{l_i}_{j=1,r\in\mathbb{R}_i}$ then
35:             return 0
36:         end if
37:     end for
38:
39:     ⑥ EVALUATING:
40:     for circuit index $r$ where $t_{r,i} = 0$ in $\tau_i = (t_{1,i}, ..., t_{s,i})$ do
41:         $y'_{i,r} \leftarrow GB.Eval(CT'_{i,r}, x'_{i,r})$
42:     end for
43:     Pick up one $y'_{i,r}$ from $\{y'_{i,r}\}_{r\notin\mathbb{R}_i}$ as the evaluating result.
44:     ⑦ After evaluating the garbled circuits $\{CT_{i,r}\}^s_{r=1}$, the verifier stores $(x'_{i,r}, y'_{i,r}, u, v)$
        into Memory $M$, where $u$ is the index of current circuit $i$, and $v$ is the index of
        the circuit which takes $y'_{i,r}$ as the input.
45: end while                    ▷ start the evaluation for the next circuit $CT_{i+1}$
46: ⑧ After evaluating all circuits, the protocol conducts the third-party verification.
47: if $VS.Eval(1^K, QA_E, QA_C, G'_c) = 0$ then
48:     return 0
49: end if
50: return 1
```

**Algorithm 5** VS.Encrypt($1^K, G_c$)

```
1: GARBLING CIRCUITS:    ▷ Generate $s$ different garbled circuits for each circuit $CT_i$
2: for all $CT_i$ in $G_c$ do
3:     for $r$ from 1 to $s$ do
4:         $(CT'_{i,r}, \{k^0_{i,j,r}, k^1_{i,j,r}\}^{n,m_i}_{i=1,j=1}) = GB.Garble(1^K, CT_i)$    ▷ Generate all wires in $CT_i$
5:     end for
6: end for
7:
8: GENERATING COMMITMENTS:
9: for all Wires which are used to encode inputs just for the inputs, in $\{CT'_{i,r}\}^{n,s}_{i=1,r=1}$, where index $j$ ranges from 1 to $l_i$ and index $r$ ranges from 1 to $s$ do
10:     Adopting the method in Definition 12, the developer generates the commitments for each wire:
11:     $\{com(k^0_{j,i,r}), com(k^1_{j,i,r})\}$
12:     Add $\{com(k^0_{j,i,r}), com(k^1_{j,i,r})\}$ to the set of commitments, $\{Com(input_i)\}^n_{i=1}$.
13: end for
14: $G'_c = [G_{struc}, \{CT'_{i,r}\}^{n,s}_{i=1,r=1}]$
15: return $G'_c$ and $\{Com(input_i)\}^n_{i=1}$ to the verifier.
```

**Algorithm 6** $VS.Encode(x_u)$
This algorithm accepts two types of inputs, the external inputs and intermediate inputs. $Type$ is used for checking the validity of the inputs.

```
1: Acquire $u, v$, which indicate the circuit indices of previous circuit and current circuit.
2: if $Type(x_u, u, v) = 1$ then                                  ▷ $x$ is an external input
3:     $x'_v \leftarrow GB.Encode(x_u, \{k^0_{u,j,r}, k^1_{u,j,r}\}^{l_u,s}_{j=1,r=1})$
4:     return $x'_v$
5: else if $Type(x_u, u, v) = 2$ then                             ▷ $x$ is an intermediate input
6:     $x'_v \leftarrow TF(x_u, \{map_{u,r}\}^s_{r=1}, \{map_{v,r}\}^s_{r=1}, u, v)$    ▷ $\{map_{u,r}\}^s_{r=1}$ and $\{map_{v,r}\}^s_{r=1}$
7:     return $x'_v$
8: end if
9: return 0
```

**Algorithm 7** VS.Checker($\tau_i$)
The developer sends the decommitments $\{dc_{i,r}\}_{r \in \mathbb{R}_i}$ to the verifier, where $dc_{i,r} = (\{\bar{k}^0_{i,j,r}, \bar{k}^1_{i,j,r}\}^{l_i}_{j=1}, \overline{sd}_i)$.

```
1: for all circuit index $r$ where $t_{r,i} = 1$ in $\tau_i = (t_{1,i}, ..., t_{s,i})$ do
2:     The developer sends the decommitments $\{dc_{i,r}\}_{r \in \mathbb{R}_i}$, which contain seed $\overline{sd}_i$ and the garbled keys $\{\bar{k}^0_{i,j,r}, \bar{k}^1_{i,j,r}\}^{l_i}_{j=1}$ to the verifier.    ▷ Where $\mathbb{R}_i$ is a set of $r$'s which satisfy $t_{r,i} \in \tau_i, t_{r,i} = 1$.
3: end for
```

**Algorithm 8** $VS.Eval(1^K, QA_E, QA_C, G'_c)$

```
1: Check the transcripts of $VS.Checker$
2: for $(Q_E, A_E) \in QA_E$ do                    ▷ check the computational result.
3:     $Q_E$ is $(x'_v, y'_v, u, v)$, $A_E$ is $\bar{x}'_v$          ▷ Involving $CT_u, CT_v$
4:     $var \leftarrow$ GB.Eval($CT'_v$, $x'_v$),
5:     if $var$ is not equal to $\bar{x}'_v$ then
6:         return 0
7:     end if
8: end for
9: for $(Q_C, A_C) \in QA_C$ do
10:    $Q_C$ is $(\tau_i)$, $A_C$ is $(\overline{sd}_i, \{\bar{k}^0_{r,j,i}, \bar{k}^1_{r,j,i}\}_{j=1}^{l_i})_{r \in \mathbb{R}_i}$.     ▷ Where $\mathbb{R}_i$ is a set of $r$'s which
   satisfy $t_{r,i} \in \tau_i, t_{r,i} = 1$
11:    Build the commitments $Com(\overline{input}_i)$ by $(\overline{sd}_i, \{\bar{k}^0_{r,j,i}, \bar{k}^1_{r,j,i}\}_{j=1}^{l_i})_{r \in \mathbb{R}_i}$.
12:    if $Com(\overline{input}_i)$ are not equal to $Com(inputi)$ in $Q_C$ then
13:        return 0
14:    end if
15: end for
16: Authorize an honest verifier, $V'$, to repeat the above verification
17: Execute $V'(1^K, G'_c, CP, EX, QA_E, QA_C)$
18: for all $CT'_i$ that $V'$ evaluates do
19:    $\overline{QA}_E, \overline{QA}_C$ are new transcripts that are generated by $V'$
20:    if $\overline{QA}_E, \overline{QA}_C$ are not equal to $QA_E, QA_C$ then
21:        return 0
22:    end if
23: end for
```

```
24: $V'$ executes verification on the specification: $(1^K, G_{spec}, EX, CP)$.
25: **if** the result is not the same as executing $G'_c$ **then**
26:     **return** 0
27: **end if**
28: **if** All the check above is run without problems **then**
29:     **return** 1
30: **end if**
```

### 3.6.2 Interaction Illustration of Our Protocol

The verifier | The developer

The developer: ① $VS.Encrypt(1^K, G_c) \rightarrow G'_c$, $\{k^0_{i,j,r}, k^1_{i,j,r}\}^{n,m_i,s}_{i,j,r=1}, Com(input_i)^n_{i=1}$

$\xleftarrow{G'_c, Com(input_i)^n_{i=1}}$

– – – – – – – – – – – – – – – – **Evaluating $\{CT'_{1,r}\}^s_{r=1}$ with $x$** – – – – – – – – – – – – – – – –

The verifier: $x = (x_1, ..., x_{l_1}), x \in EX$

$\xrightarrow{x}$

The developer: ② $VS.Encode(x, gk_1) \rightarrow \{x'_{1,r}\}^s_{r=1}$

$\xleftarrow{\{x'_{1,r}\}^s_{r=1}}$

The verifier: $\{x'_{1,r}\}^s_{r=1} = (k^{x_1}_r, ..., k^{x_{l_1}}_r)^s_{r=1}$

The developer: ③ Set the challenge string

$\tau_1 = (\tau_{1,1}, ...\tau_{1,s})$

Each $\tau_{1,r}$ is set as:

Choose $\sigma_r, seed_r$

Produce commitment $Com(\sigma_r)$

$\xleftarrow{Com(\sigma_r), seed_r}$

The verifier: Choose a random $\sigma'_r$

$\xrightarrow{\sigma'_r}$

The developer: $\tau_{1,r} = \sigma_r \oplus \sigma'_r$

$\xleftarrow{\sigma_r, sd}$

The verifier: Verify $Com(\sigma_r)$

Obtain challenge string $\tau_1$

The developer: ④ $VS.Checker(\tau_1)$

$\rightarrow \{dc_{1,r}\}_{r \in \mathbb{R}_1}$

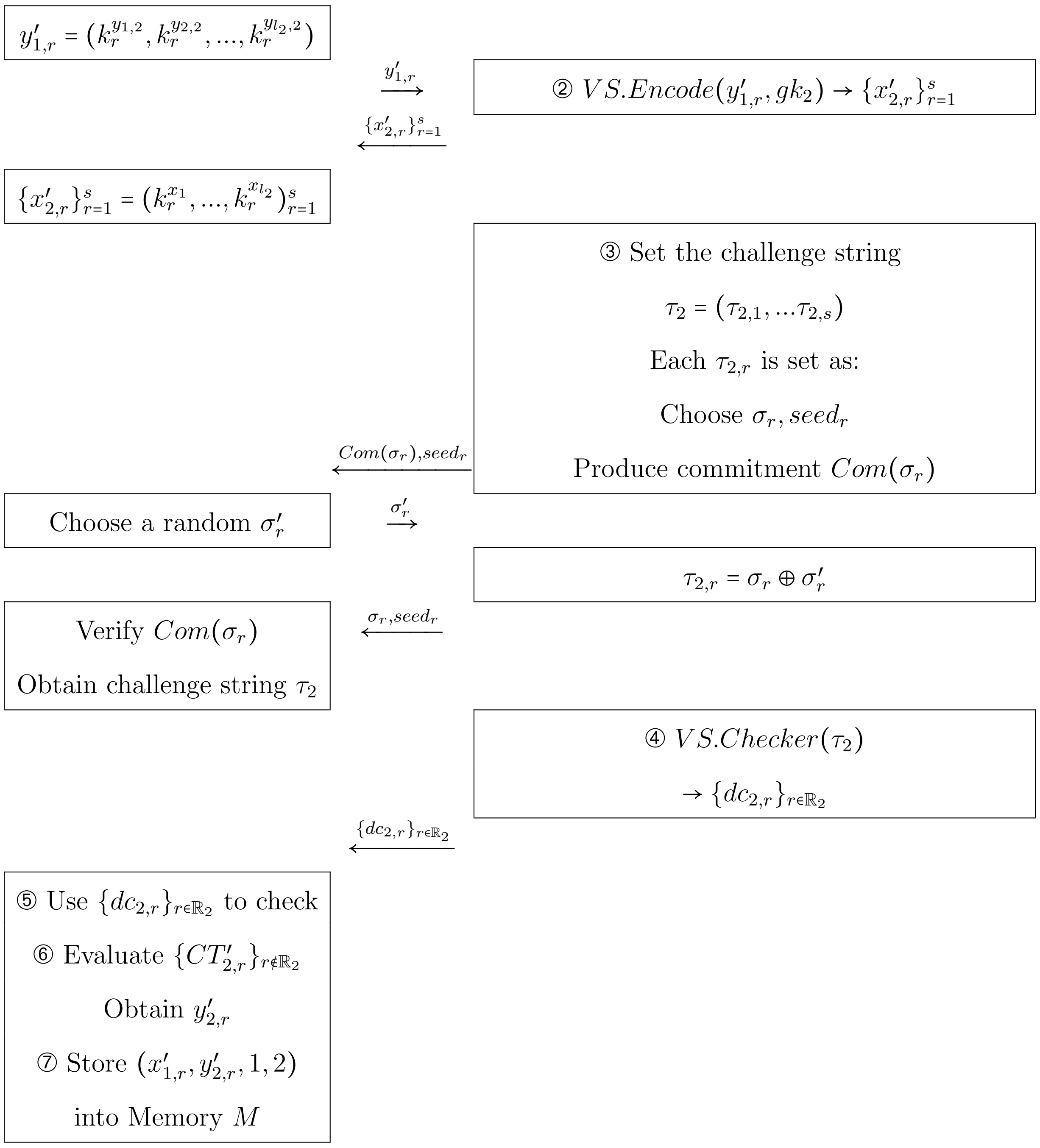
$\xleftarrow{\{dc_{1,r}\}_{r \in \mathbb{R}_1}}$

The verifier: ⑤ Use $\{dc_{1,r}\}_{r \in \mathbb{R}_1}$ to check

⑥ Evaluate $\{CT'_{1,r}\}_{r \notin \mathbb{R}_1}$,

Obtain $y'_{1,r}$

---------------**Evaluating $\{CT_{2,r}\}_{r=1}^{s}$ with $y'_{1,r}$**---------------


$y'_{1,r} = (k_r^{y_{1,2}}, k_r^{y_{2,2}}, ..., k_r^{y_{l_2,2}})$
$\xrightarrow{y'_{1,r}}$
② $VS.Encode(y'_{1,r}, gk_2) \rightarrow \{x'_{2,r}\}_{r=1}^{s}$
$\xleftarrow{\{x'_{2,r}\}_{r=1}^{s}}$
$\{x'_{2,r}\}_{r=1}^{s} = (k_r^{x_1}, ..., k_r^{x_{l_2}})_{r=1}^{s}$
③ Set the challenge string
$\tau_2 = (\tau_{2,1}, ...\tau_{2,s})$
Each $\tau_{2,r}$ is set as:
Choose $\sigma_r, seed_r$
Produce commitment $Com(\sigma_r)$
$\xleftarrow{Com(\sigma_r), seed_r}$
Choose a random $\sigma'_r$
$\xrightarrow{\sigma'_r}$
$\tau_{2,r} = \sigma_r \oplus \sigma'_r$
$\xleftarrow{\sigma_r, seed_r}$
Verify $Com(\sigma_r)$
Obtain challenge string $\tau_2$
④ $VS.Checker(\tau_2)$
$\rightarrow \{dc_{2,r}\}_{r\in\mathbb{R}_2}$
$\xleftarrow{\{dc_{2,r}\}_{r\in\mathbb{R}_2}}$
⑤ Use $\{dc_{2,r}\}_{r\in\mathbb{R}_2}$ to check
⑥ Evaluate $\{CT'_{2,r}\}_{r\notin\mathbb{R}_2}$
Obtain $y'_{2,r}$
⑦ Store $(x'_{1,r}, y'_{2,r}, 1, 2)$
into Memory $M$

– – – – – – **Repeat the process in** $\{CT_{3,r}\}_{r=1}^{s}, \{CT_{4,r}\}_{r=1}^{s}$... **with** $y'_{2,r}, y'_{3,r}, ...$ – – – – –

⋮

⋮

– – – – – – – – – – – – – – – **Evaluating** $\{CT_{n,r}\}_{r=1}^{s}$ **with** $y'_{n-1,r}$ – – – – – – – – – – – – – – –

⋮

⑥ Evaluate $\{CT'_{n,r}\}_{r\notin\mathbb{R}_n}$

Obtain $y'_{n,r}$

⑦ Store $(x'_{n,r}, y'_{n,r}, n-1, n)$

into Memory $M$

Output the decrypted result $y_n$

## 3.7 Correctness proof

We now proceed to prove if the construction in Section 3.6 satisfies Definition 15. The idea behind the proof is the comparison between real verification and ideal verification. The real verification is conducted by a real verifier and a real developer, executing an encrypted circuit graph $G'_c$, which is built as specified in our construction, with an input $x$. The honest verification is conducted by an honest verifier and an honest developer, executing an unencrypted $G_c$, which is the original circuit graph, with $x$ as well. If the results from the two kinds of verification are equal with respect to the real values, the correctness property of our protocol holds. In this case, we call $G'_c$ is equivalent to $G_c$. Here we present the formal definition.

**Definition 17.** *$CT'_r$ is a garbled circuit for circuit $CT$. $x$ is an input and its corresponding garbled input for $CT'_r$ is $x'_r$. If the result $y$ of evaluating $CT$ with $x$ is equal to the decrypted value of the result $y'_r$ of evaluating $CT'_r$ with $\{x'\}_{r=1}^{s}$, $CT$ and its corresponding encrypted circuit $\{CT'_r\}_{r=1}^{s}$ are equivalent (with respect to $x$). The equivalence is written as $CT(x) \equiv CT'_r(x'_r)$.*

*If we expand this property to a whole circuit graph, the equivalence can hold on $G_c$ and $G'_c$, i.e. $G_c(x) \equiv G'_c(x'_r)$*

This proof of correctness is expanded by induction on the number of circuits in circuit graph $G_c$.

**Base Case:** Concerning the case of evaluating a single circuit $CT$ with a given external input $x$.

The information owned by the verifier

$x$ ($x$ is an external input)

$\{CT'_r\}_{r=1}^{s}, Com(k_{r,j}^{0}, k_{r,j}^{1})_{s=1,j=1}^{s,l}, x \leftarrow VS.Encrypt(CT)$ (calling $VS.Encrypt$) (1)

$\{CT'_r\}_{r=1}^{s}, Com(k_{r,j}^{0}, k_{r,j}^{1})_{s=1,j=1}^{s,l}, \{x'_r\}_{r=1}^{s}), \leftarrow VS.Encode(x)$ (2)

where $x'_r = (k_r^{x_1}, ..., k_r^{x_l})_{r=1}^{s}$ and $l$ denotes the length of $x$. (calling $VS.Encode$)

$\{CT'_r\}_{r=1}^{s}, Com(k_{r,j}^{0}, k_{r,j}^{1})_{r=1,j=1}^{s,l}, \{x'_r\}_{r=1}^{s}, \tau, \{dc_r\}_{r\in\mathbb{R}} \leftarrow VS.Checker(\tau)$ (3)

where $\mathbb{R}$ is a set of $r$'s which satisfy $t_r \in \tau, t_{r,i} = 1$. (call $VS.Checker$)

$\Rightarrow \{CT'_r\}_{r=1}^{s}, \{x'_r\}_{r\notin\mathbb{R}}$ (decommitting and conducting consistency check) (4)

$\Rightarrow \{CT'(x'_r)\}_{r\notin\mathbb{R}} \equiv CT(x)$ (evaluate $CT'$ via $GB.Eval$) (5)

In the base case, the honest verification is run on a single circuit $CT$. Before executing the verification, the verifier has the input $x$ and the developer holds the original circuit $CT$. Thus, the result of the honest verification is $CT(x)$.

Now we observe the case of real verification. In Equation (1), the developer first invokes $VS.Encrypt$ in Algorithm 5 and obtains the garbled circuits and commitments. By the cut-and-choose strategy mentioned in Section 2.6, a circuit $CT$ has $s$ different garbled circuits. The developer sends the garbled circuits and commitments to the verifier, except the garbled keys. In Equation (2), the verifier exchanges each bit of input $x$ for the garbled key at the developer's hand via $OT$ protocol. As realized in $VS.Encode$ (Algorithm 6), the verifier obtains the encoded $x' = (k_r^{x_1}, ..., k_r^{x_l})$. In Equation (3), the two parties cooperate to set the challenge string $\tau = (t_1, ..., t_r, ..., t_s), t_r \in \{0, 1\}$. Here, $\mathbb{R}$ is denoted as a set of $r$'s which satisfy $t_r \in \tau, t_r = 1$. Thus, by the chosen decommitments from $VS.Checker$ (Algorithm 7), the verifier chooses $CT'_r$ to conduct the checks as specified in the ⑤ step of Algorithm 4, where $r \in \mathbb{R}$. In Equation (4), the verifier chooses to evaluate the circuits whose indices $r$'s satisfy $r \notin \mathbb{R}$.

With the garbled input $x' = (k_r^{x_1}, ..., k_r^{x_l})$, the garbled circuit $CT'_r$ can be computed and the result is equal to evaluate $CT$ with $x$.

**Inductive step:** First, we present the assumption for the case of $n$ circuits.

**Assumption 4.** *It is assumed that the equivalence defined in Definition 17 holds for the case of $n$ circuits.*

The Assumption 4 means that, the result of computing an unencrypted circuit graph $G_n$, consisting of $n$ circuits, with $x$ is the same as verifying its encrypted circuit graph $G'_n$ with $x$. We now proceed to prove that the result of computing an unencrypted circuit graph with $n + 1$ circuits is equivalent to the result of verifying its encrypted version.

The verification executed by an honest verifier $V'$ on the unencrypted circuit graph of $n$ circuits is denoted as:

$$V'(G_n, x) = V'(CT_1, CT_2, ..., CT_n)(x), \tag{3.3}$$

where $G_n$ represents an unencrypted circuit graph with $n$ circuits.

On the other hand, the verification run by a real verifier $V$ on the encrypted circuit graph of $n$ circuits is denoted as:

$$V(G'_n, x) = V(\{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})(x), \tag{3.4}$$

where $G'_n$ represents an encrypted circuit graph with $n$ circuits.

Then, we extend the verification to the case of $n+1$ circuits. Adding an extra circuit $CT_e$ into $G_n$ and $G'_n$, the new circuit graphs, $G_{n+1}$ and $G'_{n+1}$ have three ways to organize the circuits:

**Case 1** $CT_e$ is located as the head circuit to accept external inputs.

**Case 2** $CT_e$ is located as the end circuit to output external outputs.

**Case 3** $CT_e$ is located between two circuits, say $CT_i, CT_{i+1}$. Thus, it accepts the intermediate input from $CT_i$ and its output becomes an intermediate input for $CT_{i+1}$.

In **Case 1**, the sequences of the honest verification and the real verification are as following:

$$V'(G_{n+1}, x) = V'(CT_e, CT_1, CT_2, ..., CT_n)(x) \tag{3.5}$$

$$\Rightarrow V'(CT_1, CT_2, ..., CT_n)(CT_e(x)) \tag{3.6}$$

$$V(G'_{n+1}, x) = V(\{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})(x) \tag{3.7}$$

$$\Rightarrow V(\{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})(\{CT'_{e,r}(x)\}_{r=1}^{s}) \tag{3.8}$$

Here we make some transformations. In Equation (3.5), honest verifier $V'$ takes $x$ as the input to compute $G_n = (CT_e, CT_1, CT_2, ..., CT_n)$, which is equal to computing $(CT_1, CT_2, ..., CT_n)$ with the result of $CT_e(x)$ (in Equation (3.6)). Similarly, verifying $G'_{n+1} = (\{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})$ with $x$ in Equation (3.7) is equivalent to verifying $(\{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})$ with $(CT'_{e,r}(x))_{r=1}^{s}$ in Equation (3.8).

As the equivalence between the honest verification and the real verification holds in the $G_n, G'_n$, the equivalence between Equation (3.6) and Equation (3.6) depends on whether $CT_e(x)$ is equal to $\{CT'_{e,r}(x)\}_{r=1}^{s}$. As proved in the base case, the correctness property of our protocol holds on a single circuit. Thus, the result of computing $CT_e$ is equal to computing $CT'_{e,r}$.

But the results of $\{CT'_{e,r}(x)\}_{r=1}^{s}$ are garbled and there are $s$ different garbled outputs. To solve the first problem, we call $TF$ (Section 3.5.1) to transform the garbled results. In terms of the second problem, after evaluating $x$, $\{CT'_{e,r}(x)\}_{r=1}^{s}$ only outputs a majority value $CT'_e(x)$.

In the above ways, Equation (3.8) is equivalent to Equation (3.6). The deduction process can be shown as:

$(\{CT'_{e,r}\}_{r=1}^{s}, \underbrace{\{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}}_{\text{garbled circuit graph } G'_n}) \leftarrow VS.Encrypt(CT_e, \underbrace{CT_1, CT_2, ..., CT_n}_{\text{circuit graph } G_n})$

$(\{x'_r\}_{r=1}^{s}, \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}) \leftarrow VS.Encode(x)$

$\Rightarrow (\{CT'_{e,r}(x'_r)\}_{r\notin\mathbb{R}_e}, \{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s})$ (evaluate $\{CT'_{e,r}\}_{r=1}^{s}$)

$\Leftrightarrow (CT_e(x), \underbrace{\{CT'_{1,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}}_{\text{garbled circuit graph } G'_n})$ (by the base case on a single circuit)

$\Leftrightarrow G'_n(CT_e(x))$

$\equiv G_n(CT_e(x))$ (by Assumption 4)

In **Case 2**, the proof is quite similar to *Case 1*. Adding the extra circuit $CT_e$ as an end circuit, we still list the sequences and make some transformations:

$$V'(G_{n+1}, x) = V'(CT_1, CT_2, ..., CT_n, CT_e)(x) \tag{3.9}$$

$$\Rightarrow V'(CT_e)(G_n(x)) \tag{3.10}$$

$$V(G'_{n+1}, x) = V(\{CT'_{1,r}\}_{r=1}^{s}, \{CT'_{2,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}, \{CT'_{e,r}\}_{r=1}^{s})(x) \tag{3.11}$$

$$\Rightarrow V(\{CT'_{e,r}\}_{r=1}^{s})(G'_n(x)) \tag{3.12}$$

We can observe that the extra circuit $CT_e$ takes the output of $CT_n$ as the input, and generates the final result of $G_{n+1}$. Accordingly, in the real verification, $\{CT'_{e,r}\}_{r=1}^{s}$ is computed with the outputs of $\{CT'_{n,r}\}_{r=1}^{s}$ and generates the final result of $G'_{n+1}$. As verification on $G_n$ and $G'_n$ satisfies the correctness property, the outputs from $G_n$ and $G'_n$ are equivalent, which means the decrypted value of $G'_n(x)$ is identical to $G_n(x)$. Through $TF$, $G'_n(x)$ can be transformed into the input of $\{CT'_{e,r}\}_{r=1}^{s}$. Therefore, Equation (3.10) can be derived from Equation (3.12) as follows:

$$(\underbrace{\{CT'_{1,r}\}_{r=1}^{s},...,\{CT'_{n,r}\}_{r=1}^{s}}_{\text{garbled circuit graph } G'_n},\{CT'_{e,r}\}_{r=1}^{s}) \leftarrow VS.Encrypt(\underbrace{CT_1,CT_2,...,CT_n}_{\text{circuit graph } G_n},CT_e)$$

$$(\{x'_r\}_{r=1}^{s},\{CT'_{1,r}\}_{r=1}^{s},...,\{CT'_{n,r}\}_{r=1}^{s},\{CT'_{e,r}\}_{r=1}^{s}) \leftarrow VS.Encode(x)$$

$$\Rightarrow (\{x'_{2,r}\}_{r=1}^{s},\{CT'_{2,r}\}_{r=1}^{s},...,\{CT'_{n,r}\}_{r=1}^{s},\{CT'_{e,r}\}_{r=1}^{s}),\ \text{where } x'_{2,r} = \{CT'_{1,r}(x'_r)\}_{r\in\mathbb{R}_1}.$$

$$\vdots$$

$$\Rightarrow (\{x'_{n,r}\}_{r=1}^{s},\{CT'_{n,r}\}_{r=1}^{s},\{CT'_{e,r}\}_{r=1}^{s}),\ \text{where } x'_{n,r} = \{CT'_{n-1,r}(x'_{n-1,r})\}_{r\notin\mathbb{R}_{n-1}}.$$

$$\Rightarrow (\{CT'_{n,r}(x'_{n,r})\}_{r\notin\mathbb{R}_n},\{CT'_{e,r}\}_{r=1}^{s})$$

$$\Leftrightarrow (G'_n(x),\{CT'_{e,r}\}_{r=1}^{s})\ (G'_n(x) = (\{CT'_{1,r}\}_{r=1}^{s},...,\{CT'_{n,r}\}_{r=1}^{s})(x))$$

$$\Rightarrow \{CT'_{e,r}(G'_n(x))\}_{r\notin\mathbb{R}_e}$$

$$\Leftrightarrow \{CT'_{e,r}(G_n(x))\}_{r\notin\mathbb{R}_e}\ \text{(by Assumption 4)}$$

$$\equiv CT_e(G_n(x))\ \text{(by the base case on a single circuit)}$$

In **Case 3**, $CT_e$ appears between two circuit $CT_i$ and $CT_{i+1}$. Through the transformations of $TF$, the outputs of $\{CT'_{i,r}\}_{r=1}^{s}$ become the inputs to $\{CT'_{e,r}\}_{r=1}^{s}$ and $\{CT'_{e,r}\}_{r=1}^{s}$'s outputs become the inputs to the next circuits. The sequences of the honest verification and the real verification are as follows:

$$V'(G_{n+1},x) = V'(\underbrace{CT_1,...CT_i}_{G_i},CT_e,\underbrace{CT_{i+1}...,CT_n}_{\bar{G}_{i+1}})(x), \tag{3.13}$$

$$V(G'_{n+1},x) = V(\underbrace{\{CT'_{1,r}\}_{r=1}^{s},...,\{CT'_{i,r}\}_{r=1}^{s}}_{G'_i},\{CT'_{e,r}\}_{r=1}^{s},\underbrace{\{CT'_{n+1,r}\}_{r=1}^{s},...,\{CT'_{n,r}\}_{r=1}^{s}}_{\bar{G}'_{i+1}})(x), \tag{3.14}$$

where $G_i, G'_i$ denote the circuit graph with first $i$ circuits, while $\bar{G}_{i+1}, \bar{G}'_{i+1}$ denote the circuits starting from $(i+1)_{th}$ circuit to $n_{th}$ circuit.

$$\leftarrow VS.Encrypt(CT_1 ..., CT_i, CT_e, CT_{i+1} ... CT_n)$$

$$(\underbrace{\{CT'_{1,r}\}_{r=1}^{s} ..., \{CT'_{i,r}\}_{r=1}^{s}}_{G'_i}, \{CT'_{e,r}\}_{r=1}^{s}, \underbrace{\{CT'_{i+1,r}\}_{r=1}^{s} ..., \{CT'_{n,r}\}_{r=1}^{s}}_{\bar{G}'_{i+1}})$$

$$(\{x'_r\}_{r=1}^{s}, \{CT'_{1,r}\}_{r=1}^{s} ..., \{CT'_{i,r}\}_{r=1}^{s}, \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{i+1,r}\}_{r=1}^{s} ..., \{CT'_{n,r}\}_{r=1}^{s}) \leftarrow VS.Encode(x)$$

$$\Rightarrow (\underbrace{\{CT'_{1,r}(x'_r))\}_{r \notin \mathbb{R}_1}}_{\{x_{2,r}\}_{r=1}^{s}} ..., \{CT'_{i,r}\}_{r=1}^{s}, \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{i+1,r}\}_{r=1}^{s} ..., \{CT'_{n,r}\}_{r=1}^{s})$$

$$\Rightarrow (\underbrace{\{CT'_{2,r}(x'_{2,r})\}_{r=1}^{s}}_{\{x'_{3,r}\}_{r}^{s}}, ..., \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{i+1,r}\}_{r=1}^{s} ..., \{CT'_{n,r}\}_{r=1}^{s})$$

$$\vdots$$

Proceeding the execution to $CT_i$, the result is

$$(G'_i(\{x'_{i,r}\}_{r=1}^{s}), \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{i+1,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}). \tag{3.15}$$

The Assumption 4 can be applied here. When $n = i$, $G'_i(\{x'_{i,r}\}_{r=1}^{s})$ in Equation (3.15) is equivalent to $G_i(x_i)$. In this case, $CT_e, \{CT'_{e,r}\}_{r=1}^{s}$ are located as end circuits to $G_i, G'_i$ and we can apply the fact of **Case 2** to $G'_e = (\{CT'_{1,r}\}_{r=1}^{s} ..., \{CT'_{e,r}\}_{r=1}^{s})$ and $G'_e = (CT_1 ..., CT_i)$here

$$(G'_i(\{x'_{i,r}\}_{r=1}^{s}), \{CT'_{e,r}\}_{r=1}^{s}, \{CT'_{i+1,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}) \tag{3.15}$$

$$\Rightarrow (G'_e(\{x'_{e,r}\}_{r=1}^{s}), \{CT'_{i+1,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}) \tag{3.16}$$

$$\Leftrightarrow (G_e(x_e), \{CT'_{i+1,r}\}_{r=1}^{s}, ..., \{CT'_{n,r}\}_{r=1}^{s}) \text{ (by the conclusion of Case 2)} \tag{3.17}$$

This fact indicates that circuit graph $G'_e(\{x'_{e,r}\}_{r=1}^{s})$ is equivalent to $G_e(x_e)$. Consequently, $\bar{G}_{i+1}, \bar{G}'_{i+1}$ accept the equivalent results $G_e(x), G'_e(x)$, and thus the task of proving the equivalence between $G_{n+1}$ and $G'_{n+1}$ becomes the one of proving the equivalence between $\bar{G}_{i+1}, \bar{G}'_{i+1}$.

Repeatedly using the conclusion of **Case 2**, we can make circuit-by-circuit derivations on Equation (3.17):

$$(\underbrace{G'_e(\{x'_{e,r}\}_{r=1}^s)}_{\{x'_{i'+1,r}\}_{r=1}^s}, \{CT'_{i'+1,r}\}_{r=1}^s, ..., \{CT'_{n',r}\}_{r=1}^s) \tag{3.16}$$

$$\Rightarrow (\underbrace{G'_{i+2}(\{x'_{i'+1,r}\}_{r=1}^s)}_{\{x'_{i'+2,r}\}_{r=1}^s}, \underbrace{\{CT'_{i'+2,r}\}_{r=1}^s, ..., \{CT'_{n',r}\}_{r=1}^s}_{\bar{G}'_{i'+2}}), \tag{3.18}$$

$$\Leftrightarrow (G_{i+2}(x_{i'+1}), \underbrace{\{CT'_{i'+2,r}\}_{r=1}^s, ..., \{CT'_{n',r}}_{\bar{G}_{i'+2}}\}_{r=1}^s) \text{ (by the conclusion of Case 2)} \tag{3.19}$$

where $i'$ denotes the original circuit index in $G'_n, G_n$ to distinguish from the $i$ in $G'_{n+1}, G_{n+1}$. Similarly, $n'$ denotes the original total number of circuits before adding $CT_e$.

Based on the equivalence between $G'_e(\{x'_{e,r}\}_{r=1}^s)$ is equivalent to $G_e(x_e)$, $\{CT'_{i'+1,r}\}_{r=1}^s$ and $CT_{i'+1}$ can be regarded as end circuits to $G'_{i+2} = (\{CT'_{1,r}\}_{r=1}^s..., \{CT'_{e,r}\}_{r=1}^s, \{CT'_{i'+1,r}\}_{r=1}^s)$ and $G_{i+2} = (CT_1..., CT_e, CT_i)$. So Equation (3.18) and (3.19) are proved to beequivalent as well.

This process can be repeated to $\{CT'_{n'-1,r}\}_{r=1}^s$ and $CT_{n'-1}$.

$$\underbrace{(G'_n(\{x'_{n'-1,r}\}_{r=1}^s)}_{\{x'_{n',r}\}_{r=1}^s}, \underbrace{\{CT'_{n',r}\}_{r=1}^s}_{\bar{G}'_{n'}}) \text{ (extend the equivalence to the case of } n'-1 \text{ circuits)}$$

$$\Rightarrow \{CT'_{n',r}(x'_{n',r})\}_{r \notin \mathbb{R}_{n'}} \text{ (}\{CT'_{n',r}\}_{r=1}^s \text{ and } CT_{n'} \text{ are end circuits to } G'_{n+1}, G_{n+1})$$

$$\Leftrightarrow G'_{n+1}(\{x'_{n',r}\}_{r=1}^s)$$

$$\equiv G_{n+1}(x_{n'}) \text{ (by the conclusion of Case 2)}$$

By the above calculations, the honest verification on $G_{n+1}$ is equivalent to the real verification on $G'_{n+1}$ in **Case 3**. Combined with the results in **Case 1** and **Case 2**, the inductive step is also proved. Finally, the correctness property of the protocol holds. □

## 3.8 Security proof for malicious case

Our security is based on the *simulation paradigm* (Goldreich, 2006, Section 4.3.1). A protocol in this paradigm is secure *if a simulated party only has access to the input and output of an original party, he can still manage to simulate any behavior that this party may conduct in this protocol.* This definition implies that the behavior of parties is so constrained by the protocol that any information, which the party can acquire, cannot goes beyond the scope of the known information.

To realize the simulation, we design experiments to generate the indistinguishable distribution ensembles. The *view* of a party is all the information that the party has received and owned in a certain procedure. What kind of information is private has been defined in Section 2.2 and 3.1. The work in (Goldreich, 2004, 2006) and (Lindell and Pinkas, 2009) includes more detailed explanation about these concepts.

The definition of security has been provided in Definition 16, which involves a real experiment and an ideal experiment. The real experiment is actually what the protocol runs on an encrypted circuit graph $G'_c$. The ideal experiment is a virtual procedure to simulate the real experiment without giving access to private information. Thus, the ideal experiment needs three simulators $S_1, S_2, S_3$ to simulate the behavior of $VS.Encrypt, VS.Encode$ and $VS.Checker$ respectively. As mentioned in Section

1.1, the protocol allows leaking some extra information to the verifier during verification, like the structure graph $G_{struc}$. For this reason, the parties in ideal experiment should also have access to the extra information. This job is performed by *Oracles*, which can be regarded as "black boxes" that are able to respond the answers for the queries from a party.

The idea of this proof is to realize the indistinguishability between the real experiment and the ideal experiment, which are specified in Definition 16. The security holds if the view in the real experiment is indistinguishable with the view in the ideal experiment. Given test cases as external inputs, we execute these experiments on the whole circuit graph and check if the views in these experiments are indistinguishable. The process of executing the experiments is similar to the process of evaluation (defined in Section 2.2) – that is, the process of running the experiments can be viewed as the repeated computation on each circuit. We provide the proof in two cases: running the experiments with external inputs or intermediate inputs.

### 3.8.1 Simulators in the Case of External Inputs

The execution starts from evaluating external inputs. It is assumed that the index of first circuit is 1 and the last circuit is $n$. As defined in Section 2.2, $IC$ denotes the set of circuits accepting external inputs. Therefore, for a circuit $CT_i \in IC$, we build the simulators $S_1, S_2, S_3$ in ideal experiment in Definition 16 as following:

Build $S_1$:

1. Take as input circuit structure $G_{struc}$.
2. Choose a different pseudorandom generator $\widetilde{Gr}$ from $Gr$ in the real experiment.

$\widetilde{Gr}$ is used to generate commitments.

3. Build garbled circuits $\{\widetilde{CT}'_{i,r}\}_{i=1,r=1}^{n,s}$ and garbled keys $\{\tilde{k}_{i,j,r}^0, \tilde{k}_{i,j,r}^1\}_{i=1,j=1,r=1}^{n,m_i,s}$.

4. Generate commitments for the garbled keys which are used to encode inputs: $\{Com(\widetilde{input}_i)\}_{i=1}^n = \{Com(\widetilde{gk}_i)\}_{i=1}^n$, where $\widetilde{gk}_i = \{\tilde{k}_{i,j,r}^0, \tilde{k}_{i,j,r}^1\}_{j=1,r=1}^{l_i,s}$.

Build $S_2$:

1. Take an external input $x \in EX$ and garbled keys in the real experiment $\{\tilde{k}_{j,r}^0, \tilde{k}_{j,r}^1\}_{j=1,r=1}^{l,s}$ as the inputs.

2. Given access to oracle $O_1$, $S_2$ obtains the result of $Type(x)$.

3. Compute the result $GB.Encode(x, \{\tilde{k}_{j,r}^0, \tilde{k}_{j,r}^1\}_{j=1,r=1}^{l,s})$.

Build $S_3$:

1. Take a challenge string $\tau_i$ in the real experiment as the input.

2. Given access to oracle $O_2$, $S_3$ obtains the seed $sd_i$ that participates the coin-tossing protocol to generate the challenge string $\tau_i$. $S_3$ fetches the garbled keys for inputs, $\{\tilde{k}_{i,j,r}^0, \tilde{k}_{i,j,r}^1\}_{i=1,j=1,r=1}^{n,l_i,s}$, from $S_1$.

3. Return $sd_i$ and $\{\tilde{k}_{i,j,r}^0, \tilde{k}_{i,j,r}^1\}_{j=1,r=1}^{l_i,s}$

Now we can present the unfinished views of real experiment and ideal experiment when the experiments accept external inputs:

$$\begin{aligned} View_{real} &= (G, CP, state_A, G_{struc}, \{CT'_{i,r}\}_{i=1,r=1}^{n,s}, \{Com(input_i)\}_{i=1}^{n}, x, \{x'_r\}_{r=1}^{s} \\ &\quad \{k_{j,i,r}^0, k_{j,i,r}^1\}_{j=1,r=1}^{l_i,s}, sd_i, \tau_i, ...) \qquad (3.20) \\ View_{ideal} &= (G, CP, state_A, G_{struc}, \{\widetilde{CT}'_{i,r}\}_{i=1,r=1}^{n,s}, \{Com(\widetilde{input}_i)\}_{i=1}^{n}, x, \\ &\quad GB.Encode(x, \widetilde{gk}_1\}), \{\tilde{k}_{j,i,r}^0, \tilde{k}_{j,i,r}^1\}_{j=1,r=1}^{l_i,s}, sd_i, \tau_i, ...) \qquad (3.21) \end{aligned}$$

### 3.8.2 Simulators in the Case of Intermediate Inputs

The above views are incomplete because only the circuits accepting external inputs are considered. Now we discuss the case when circuits accept intermediate inputs, which derived from previous circuits.

The simulators in the case of intermediate inputs are slightly different. $S_1$ is not invoked any more after generating garbled circuits and garbled keys. As the $VS.Encode$ adopts different strategies to cope with intermediate inputs, the way of building $S_2$ is changed accordingly. The way to build $S_3$ is still the same as above.

Now we need to rebuild $S_2$ to simulate the views in the real experiment. In this case, intermediate outputs $\{y'_{i,r}\}_{i=1}^{n-1}$ and inputs $\{x'_{i,r}\}_{i=2,r=1}^{n,s}$ are confidential to the developer. Particularly, $\{y'_{i,r}\}_{i=1}^{n-1}$ only have $n-1$ values, while $\{x'_{i,r}\}_{i=2,r=1}^{n,s}$ have $(n-1) \cdot s$ values. Thus, $S_2$ cannot get access to $\{y'_{i,r}\}_{i=1}^{n-1}$ and $\{x'_{i,r}\}_{i=2,r=1}^{n,s}$ as well. In order to simulate the intermediate inputs $\{y'_i\}_{i=1}^{n-1}$, $S_2$ has to make use of the simulator $S_{GB}$ for garbled circuits (see Definition 9) and the oracle $O_1$.

Let us review how $TF$ works. In the function $TF$, the developer holds the input $\{map_{i-1,r}\}_{r=1}^{s}, \{map_{i,r}\}_{r=1}^{s}$. Similarly, $S_2$ can also own $\{map_{i-1,r}\}_{r=1}^{s}, \{map_{i,r}\}_{r=1}^{s}$. In real experiment, $\{x'_{i,r}\}_{r=1}^{s}$ are the inputs to the current garbled circuits $\{CT'_{i,r}\}_{r=1}^{s}$. But at the meantime, $\{x'_{i,r}\}_{r=1}^{s}$ are also the actual outputs of computing $TF$. As explained in Section 3.5.1, running $TF$ is actually running $\Gamma_{compare}$. In the garbled circuit $\Gamma_{compare}$, $\{y'_{i,r}\}_{i=1}^{n-1}$ are the inputs from the verifier, while $\{map_{i-1,r}\}_{r=1}^{s}, \{map_{i,r}\}_{r=1}^{s}$ are the inputs from the developer. $\{x'_{i,r}\}_{r=1}^{s}$ is the output from computing $\Gamma_{compare}$ and is returned to the verifier.

What $S_2$ tries do is to simulate the intermediate input $y'_{i-1,r}$ from the verifier, where $i-1$ indicates that $y'_{i-1,r}$ is the result from a previous circuit. Here $S_2$ needs to cooperate with oracle $O_1$, but he cannot request to acquire the confidential information $\{x'_{i,r}\}_{r=1}^{s}$ from $O_1$. As $O_1$ has access to the garbled circuit $\Gamma_{compare}$, $O_1$ can compute $\Gamma_{compare}$ with the inputs provided by $S_2$. Therefore, $S_2$ provides the mapping tuples $\{map_{i-1,r}\}_{r=1}^{s}, \{map_{i,r}\}_{r=1}^{s}$ to oracle $O_1$, and the latter runs the simulator of garbled circuits $S_{GB}$ for $\Gamma_{compare}$. According to Definition 9, $S_{GB}$ can manage to simulate the input from the verifier by the developer's input $\{map_{i-1,r}\}_{r=1}^{s}, \{map_{i,r}\}_{r=1}^{s}$ and the verifier's output $\{x'_{i,r}\}_{r=1}^{s}$, as well as the size of target garbled circuit $1^{|\Gamma_{compare}|}$ and target input $1^{|y'_{i-1,r}|}$. In this way, a simulated $\tilde{y}'_{i-1,r}$ is generated.

In the meantime, $S_2$ can take as input the simulated $y'_{i-1,r}$ to compute the simulated $\{\tilde{x}'_{i,r}\}_{i=1}^{s}$. For circuit $CT_i \in G_c$ and $CT_i \notin IC$, $CT_i$'s input derives from circuit $CT_{i-1}$. So the construction for simulator $S_2$ is such as:

Build $S_2$ for intermediate inputs:

1. Take as input the mapping tuples $\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}$.
2. Build a garbled circuit $\tilde{\Gamma}_{compare}$ as specified in Algorithm 2 and obtain the garbled keys $\widetilde{gk}_{compare}$.
3. Given access to oracle $O_1$, $S_2$ obtains the result of $Type(x)$ and requests $O_1$

to compute $\tilde{y}'_{i-1} = S_{GB}(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, \{x'_{i,r}\}_{r=1}^{s}, 1^{|\Gamma_{compare}|}, 1^{|y'_{i-1,r}|})$ (Definition 9).

4. Then, $S_2$ computes $\{\tilde{x}'_i\}_{r=1}^{s} = TF(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, \tilde{y}'_{i-1}, i-1, i)$.

5. Return $\tilde{y}'_{i-1}$ and $\{\tilde{x}'_i\}_{r=1}^{s}$.

Overall, combined with the views in Equation (3.20) and (3.21), the complete views for two experiments are such as:

$$\begin{aligned}
View_{real} = (&G, CP, state_A, G_{struc}, \{CT'_{i,r}\}_{i=1,r=1}^{n,s}, x, x',\\
&\{y'_{i,r}\}_{i=1,r\notin\mathbb{R}_i}^{n-1}, \{x'_{i,r}\}_{i=2,r=1}^{n,s}, \{dc_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}, \{\{Type(x'_{i,r})\}_{r=1}^{s}, \tau_i\}_{i=1}^{n},\\
&\{k_{j,i,r}^{0}, k_{j,i,r}^{1}\}_{i=1,j=1,r=1}^{n,l_i,s}, \{Com(input_i)\}_{i=1}^{n}), \qquad (3.22)\\
View_{ideal} = (&G, CP, state_A, G_{struc}, \{\widetilde{CT}'_{i,r}\}_{i=1,r=1}^{n,s}, x, GB.Encode(x, \widetilde{gk}_1),\\
&\{\tilde{y}'_{i,r}\}_{i=1,r\notin\mathbb{R}_i}^{n-1}, \{\tilde{x}'_{i,r}\}_{i=2,r=1}^{n,s}, \{\widetilde{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}, \{\{Type(x'_{i,r})\}_{r=1}^{s}, \tau_i\}_{i=1}^{n},\\
&\{\tilde{k}_{j,i,r}^{0}, \tilde{k}_{j,i,r}^{1}\}_{j=1,r=1}^{n,l_i,s}, \{Com(\widetilde{input}_i)\}_{i=1}^{n}), \qquad (3.23)
\end{aligned}$$

*where $\mathbb{R}_i$ is a set of r's which satisfy $t_{r,i} \in \tau_i, t_{r,i} = 1$*

## 3.8.3 Proving Security

Now we can start proving the indistinguishability between real experiment $Exp_{real}$ and ideal experiment $Exp_{ideal}$. First, we build a hybrid experiment $Exp_{HE-1}$ to assist the proof.

*Note:* In the hybrid experiment, the garbled keys outputted by $VS.Encrypt()$ are different.

| $Exp_{HE_1}(1^K)$ |
|---|
| 1. $(G, CP, state_A) \leftarrow A_1(1^K)$ |
| 2. $\hat{G}' \leftarrow VS.Encrypt(1^K)$ |
| 3. $a \leftarrow A_2^{S_2^{O_1}, S_3^{O_2}}(1^K, \hat{G}', G, CP, state_A)$ |
| 4. Output $a$ |

Table 3.7: Hybrid experiment $Exp_{HE\text{-}1}$

The hybrid experiment $Exp_{HE\text{-}1}$ derives the the simulators $S_2, S_3$ from the ideal experiment, but it also calls $VS.Encrypt$ of the real experiment. Because the complete sets of garbled keys are also private information, the developer in $Exp_{HE\text{-}1}$ has to generate different garbled keys. This is simply done by invoking pseudorandom generator twice, and the new pseudorandom strings would be indistinguishable with the garbled keys in the real experiment. Accordingly, the commitments in $Exp_{HE\text{-}1}$ are different but indistinguishable with the commitments in the real experiment. $S_2, S_3$ use the above garbled keys and follow the construction in Section 3.8.1 and 3.8.2. Thus, the view of hybrid experiment is:

$$\begin{aligned} View_{HE\text{-}1} = (&G, CP, state_A, G_{struc}, \{\widehat{CT}'_{i,r}\}_{i=1,r=1}^{n,s}, x, GB.Encode(x, \widehat{gk}_1), \\ &\{\hat{y}'_{i,r}\}_{i=1,r\notin\mathbb{R}_i}^{n-1}, \{\widehat{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}, \{\hat{x}'_i\}_{i=2}^{n}, \{\{Type(x'_{i,r})\}_{r=1}^{s}, \tau_i\}_{i=1}^{n}, \\ &\{\hat{k}_{j,i,r}^0, \hat{k}_{j,i,r}^1\}_{j=1,r=1}^{n,l_i,s}, \{Com(\widehat{input}_i)\}_{i=1}^{n}) \end{aligned} \tag{3.24}$$

*where* $\mathbb{R}_i$ *is a set of r's which satisfy* $t_{r,i} \in \tau_i, t_{r,i} = 1$

**The indistinguishability between $Exp_{ideal}$ and $Exp_{HE\text{-}1}$.** It is obvious to achieve the indistinguishability between the ideal experiment and the hybrid experiment. The garbled circuits in $Exp_{ideal}$ are built with the same structure as the ones in $Exp_{HE\text{-}1}$. As described in Section 2.4.2, a garbled circuit consists of garbled truth tables, which are composed of garbled keys for each wire. As the garbled keys in $Exp_{HE-1}$ and $Exp_{ideal}$ are generated by the same pseudorandom generator, $\{\hat{k}^0_{j,i,r}, \hat{k}^1_{j,i,r}\}^{n,l_i,s}_{j=1,r=1}$ in $Exp_{HE-1}$ are indistinguishable with $\{\tilde{k}^0_{j,i,r}, \tilde{k}^1_{j,i,r}\}^{n,l_i,s}_{j=1,r=1}$. This fact leads to the indistinguishability of the construction on this basis. For this reason, the garbled circuits and commitments in the hybrid experiment are indistinguishable with the ones in ideal experiment.

$S_2, S_3$ in the hybrid experiment generate the similar view as the ideal experiment, regardless of the cases of external inputs or intermediate inputs. So the view of the hybrid experiment is computationally indistinguishable with the view of the ideal experiment.

**The indistinguishability between $Exp_{real}$ and $Exp_{HE-1}$.** Next, we need to prove the view of the hybrid experiment is also computationally indistinguishable with the view of the real experiment. The proof is by contradiction.

It is assumed that a distinguisher $D$ can distinguish the views of $Exp_{HE-1}$ and $Exp_{real}$. Thus, there is a polynomial $p(\cdot)$ such that for infinitely many $K$,

$$\left|Pr[D(Exp_{real}(1^K)) = 1] - Pr[D(Exp_{HE-1}(1^K)) = 1]\right| \geq \frac{1}{p(K)} \tag{3.25}$$

We will use such a distinguisher $D$ to challenge the existing assumptions. The views of of $View_{HE-1}$ and $View_{real}$ can be categorized by the message sources.

Table 3.8 presents the comparison of views between the real experiment and the hybrid experiment. Except the shared information, the messages in the hybrid experiment are different from the ones in the real experiment. Thus, if distinguisher $D$ detects the distinctions between the two views, the distinctions must be fell in one of the four categories.

Firstly, we assume that $D$ can distinguish the messages in *Row 2*, Table 3.8. As

| | Real experiment | Hybrid experiment |
|---|---|---|
| *Row 1* Shared information | $G, CP, state_A, G_{struc}, \{\tau_i\}_{i=1}^{n}$ | $G, CP, state_A, G_{struc}, \{\tau_i\}_{i=1}^{n}$ |
| *Row 2* Garbling circuits | When invoking $VS.Encrypt$: $(\{CT'_{i,r}\}_{i=1,r=1}^{n,s}, \{Com(input_i)\}_{i=1}^{n}, \{k_{j,i,r}^{0}, k_{j,i,r}^{1}\}_{i=1,j=1,r=1}^{n,l_i,s})$ | When invoking $VS.Encrypt$: $(\{\widehat{CT}'_{i,r}\}_{i=1,r=1}^{n,s}, \{Com(\widehat{input_i})\}_{i=1}^{n}, \{\hat{k}_{j,i,r}^{0}, \hat{k}_{j,i,r}^{1}\}_{i=1,j=1,r=1}^{n,l_i,s})$ |
| *Row 3* Encoding external inputs | When invoking $VS.Encode$: $(x, Type(x), x')$ | When invoking $S_2^{O_1}$ : $(x, Type(x), GB.Encode(x, \widehat{gk}_1)$ |
| *Row 4* Encoding intermediate inputs and evaluating | When invoking $VS.Encode$: $(\{y'_{i,r}\}_{i=1,r\notin\mathbb{R}_i}^{n-1}, \{x'_{i,r}\}_{i=2,r=1}^{n,s}, \{Type(x'_{i,r})\}_{i=2,r=1}^{n,s})$ | When invoking $S_2^{O_1}$ : $(\{\hat{y}'_{i,r}\}_{i=1,r\notin\mathbb{R}_i}^{n-1}, \{\hat{x}'_{i,r}\}_{i=2,r=1}^{n,s}, \{Type(x'_{i,r})\}_{i=2,r=1}^{n,s})$ |
| *Row 5* Obtaining decommitments | When invoking $VS.Checker$: $\{dc_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}$ | When invoking $S_3^{O_2}$ : $\{\widehat{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}$ |

Table 3.8: Decomposition of the views of $View_{HE-1}$ and $View_{real}$

mentioned earlier, the hybrid experiment follows the same ways of building the garbled circuits and commitments as specified in Algorithm 5, except generating different garbled keys. Consequently, the fact that $D$ can distinguish the messages of *Row 2* leads to the fact that $D$ can distinguish the garbled keys, which are just pseudorandom strings. Obviously, this assumption contradicts the security of pseudorandom generators.

Secondly, we assume that $D$ can distinguish the messages in *Row 3*, Table 3.8. *Row 3* indicates the information which is generated in the phase of computing external inputs. In this phase, the developer in experiment $Exp_{real}$ calls $VS.Encode$ (Algorithm 6), while in experiment $Exp_{HE-1}$ he calls the simulator $S_2$. $x$ is an external input and it is publicly known. $Type(x)$ is the result of running function $Type$ (Algorithm 3) with $x$. As $Type(x)$ is not private information, $S_2$ in the hybrid experiment can request it from oracle $O_1$. In this case, the information distinguished by $D$ is $x'$ and $GB.Encode(x, \widehat{gk}_1)$.

In fact, $x'$ is computed by $GB.Encode(x, gk_1)$ in the real experiment. The difference $GB.Encode(x, \widehat{gk}_1)$ is using different garbled keys. However, distinguishing garbled keys conflicts with the security of pseudorandom generator. For this reason, the assumption that $D$ can distinguish the messages of *Row 3* contradicts.

Thirdly, we assume that $D$ can distinguish the messages in *Row 4*, Table 3.8. The information in *Row 4* is produced during encoding intermediate inputs. First of all, $Type(x'_i)$ is shared by the both parties, so it cannot be distinguished.

Then, we assume that $D$ can distinguish $\hat{y}'_{i-1,r} = S_{GB}(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, \{x'_{i,r}\}_{r=1}^{s}, 1^{|\Gamma_{compare}|}, 1^{|y'_{i-1,r}|})$ and $y'_{i-1,r} \in \{y'_{i-1,r}\}_{i=2,r\notin\mathbb{R}_i}^{n} = \{CT'_{i,r}(x'_{i-1,r})\}_{i=2,r\notin\mathbb{R}_i}^{n}$. By Definition 9, the simulator of garbled circuits $S_{GB}$ can generate a party's view by the computational result $C(x)$, the input of the party, along with the size of the circuit and input $1^{|C|}, 1^{|x|}$. In our case, $\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}$ are the inputs owned by the developer, $x'_{i,r}$ is the computation result of $\Gamma_{compare}$ (Algorithm 2), and $1^{|\Gamma_{compare}|}, 1^{|y'_{i-1,r}|}$ denote the size of garbled circuit $\Gamma_{compare}$ and input $y'_{i-1}$ respectively. With these values, invoking $S_{GB}(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, x'_{i,r}, 1^{|\Gamma_{compare}|}, 1^{|y'_{i-1,r}|})$ can generate a simulated input for the verifier, which is exactly $\hat{y}'_{i-1,r}$.

Therefore, by the security of garbled circuits, $y'_{i-1,r}$ are indistinguishable from $\hat{y}'_{i-1,r} = S_{GB}(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, x'_{i,r}, 1^{|\Gamma_{compare}|}, 1^{|x'_{i-1,r}|})$, which contradicts the assumption that $D$ can distinguish the two values.

Subsequently, the assumption that $D$ can distinguish $\{x'_{i,r}\}_{r=1}^{s}$ and $\{\hat{x}'_{i,r}\}_{r=1}^{s} = TF(\{map_{i-1,r}, map_{i,r}\}_{r=1}^{s}, \hat{y}'_{i-1,r}, i-1, i)$, which can be also proved as contradiction. Basically, $\{x'_{i,r}\}_{r=1}^{s}$ or $\{\hat{x}'_{i,r}\}_{r=1}^{s}$ are generated from $y'_{i-1,r}$ and $\hat{y}'_{i-1,r}$. As $\{y'_{i-1,r}\}_{i=2,r\notin\mathbb{R}_{i-1}}^{n}$

in the real experiment have been proved to be indistinguishable with $\{\hat{y}'_{i-1,r}\}_{i=2,r\notin\mathbb{R}_{i-1}}^{n}$ in the hybrid experiment, it leads to the fact that $\{x'_{i,r}\}_{r=1}^{s}$ or $\{\hat{x}'_{i,r}\}_{r=1}^{s}$ are also indistinguishable.

Lastly, we assume that $D$ can distinguish the messages in *Row 5* Table 3.8. In the real experiment, the developer calls $VS.Checker$ and returns the decommitments $\{dc_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n} = (sd_i, \{k_{j,i,r}^{0}, k_{j,i,r}^{1}\}_{j=1}^{l_i})_{i=1,r\in\mathbb{R}_i}^{n}$. In the hybrid experiment, the developer calls the simulator $S_3$ and obtains $\{\widehat{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n} = (\widehat{sd}_i, \{\hat{k}_{j,i,r}^{0}, \hat{k}_{j,i,r}^{1}\}_{j=1}^{l_i})_{i=1,r\in\mathbb{R}_i}^{n}$. Obviously, the distinctions between $\{dc_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}$ and $\{\widehat{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}$ are simply the garbled keys. As the previous case, the assumption that $D$ distinguishes $\{dc_{i,r}\}_{i=1,r\in mathbbR_i}^{n}$ from $\{\widehat{dc}_{i,r}\}_{i=1,r\in\mathbb{R}_i}^{n}$ contradicts the security of pseudorandom generator.

Therefore, the view in real experiment is indistinguishable with the view in the hybrid experiment. As the indistinguishability has been proved between the views of the ideal experiment and hybrid experiment, we finally achieve that the view in ideal experiment is indistinguishable with the view in real experiment. $\square$

# Chapter 4

# Conclusion

Our work proposes a protocol, based on garbled circuits, to realizes the secure and trusted “partial white-box verification”. With a certain leakage of information, like, revealing the structure of the program, the protocol satisfies the requirements about security and correctness. Particularly, through a series of consistency checks or validity checks, the protocol can efficiently defend against the malicious behavior.

However, this work is not the end of the research about this subject. The updates for existing techniques or new techniques for existing problems may bring better solutions, like the possible advance in implementing $FHE$ and *verifiable computation*. Or the changes about the security requirements may pose new questions, like how to hide the graph structure. Here we present some subjects as our future work.

**Comparison with $FHE$'s construction.** As our work derived from the protocol based on $FHE$ (Cai *et al.*, 2016), comparison between the two protocols will be the most possible job in the next step. The comparison may focus on the implementation difficulty and computational efficiency. Although the current protocol based on garbled circuits has been implemented in (Ji, 2016), the implementation of the

$FHE$'s version is still missing. This part of job will be supplied as the next plan.

**Computational efficiency and optimization.** Improving the efficiency of our protocol has much work to do. Since the birth of garbled circuits, the optimization around the technique has never ended. Usually, people care about three aspects that affect the computational efficiency of the protocol of garbled circuits: *the communication cost, the execution cost and the circuit cost* (Snyder, 2014).

The execution cost and the circuit cost mainly rely on the size of the circuit graph $G_c$, for example, the number of circuits, the number of gates in each circuit, and the security parameter $s$ etc. The work about such optimization can bee seen in (Pinkas *et al.*, 2009; Malkhi *et al.*, 2004; Huang *et al.*, 2011; Kreuter *et al.*, 2012).

Specifically, we need to pay more attention to the communication cost. The number of rounds of executing oblivious transfer is the key factor in this kind of cost. If the protocol does not adopt the cut-and-choose strategy, encoding an input with $n$ bits needs $n$ rounds of $OT$ protocol, which can only exchange one bit in one round (Section 2.5). If the cut-and-choose approach is applied to the protocol, the number of inputs that need be encoded increase to $n \cdot s$. Accordingly, encoding the inputs corresponding to a same circuit generates $n \cdot s$ rounds of $OT$ protocol. On the other hand, a feature in our protocol is to realize the function $TF$ to securely encode the intermediate inputs. However, as presented in Section 3.5.1, this method also brings a rather big communication cost. In the garbled circuit $\Gamma_{compare}$, the input from the verifier is a garbled value $\{y'_{i,r}\} = (y_r^{k_{i,1}}, ..., y_r^{k_{i,l_i}})$. If the length of a garbled key is $p$ and $\{y'_{i,r}\}$ has $l_i$ garbled keys, the total bits in such a $\{y'_{i,r}\}$ are $p \cdot l_i$. Consequently, the rounds of $OT$ that are invoked may reach $s \cdot p \cdot l_i$.

Discussed in (Lindell and Pinkas, 2007), this paper presented a probabilistic construction to reduce the wires of input, which also reduces the number of rounds of calling $OT$. The related work can be added to the implementation of our protocol (Ji, 2016) and optimize its computation.

**Limitations of garbled circuits.** A key limitation of garbled circuit is that a set of garbled circuits and the garbled keys can only be used once. This problem severely restrains the application scope of garbled circuits. Some work has been invested in implementing the reusable garbled circuits, like the attempts made by Goldwasser et al. in (Goldwasser *et al.*, 2013), which is baed on $FHE$. If this defect about garbled circuits can be solved in the future, the application scenarios of garbled circuits, as well as our protocol, would be much broader.

**Verifiable computation.** We have mentioned *verifiable computation* in section 1.2. Although $VC$ is not contained in our construction, the discussion about this technique is still valuable to our subject. In the work of (Cai *et al.*, 2016), the author made use of $FHE$ to reverse the computation and realized checking the validity of $VS.Encode$'s results. However, this approach actually repeats the computation. As $FHE$ is "notorious" for its low computational efficiency, this kind of checks is not acceptable. Alternatively, $VC$ provides the more precise and efficient notions to conduct such checks. The *probabilistically checkable proofs* ($PCP$), harwired in $VC$, provides a good hint. As the $PCP$ theorem implies that an assertion can be checked effectively by a proof in only a few of bits (Arora and Safra, 1998), this possible approach would significantly save the cost when checking the computational results. When the research on practical $VC$ takes the essential breakthrough, checking the results of $VS.Encode$ would have better solutions.

# Bibliography

Abadi, M. and Feigenbaum, J. (1990). Secure circuit evaluation. *Journal of Cryptology*, **2**(1), 1–12.

Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker, R. A., and Parnas, D. L. (1992). Software requirements for the a-7e aircraft. Technical report, DTIC Document.

Arora, S. and Safra, S. (1998). Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM (JACM)*, **45**(1), 70–122.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im) possibility of obfuscating programs. In *Advances in cryptology-CRYPTO 2001*, pages 1–18. Springer.

Bellare, M. and Micali, S. (1989). Non-interactive oblivious transfer and applications. In *Advances in Cryptology-CRYPTO'89 Proceedings*, pages 547–557. Springer.

Bellare, M., Hoang, V. T., and Rogaway, P. (2012). Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM.

Ben-David, A., Nisan, N., and Pinkas, B. (2008). Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM.

Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. (2013). Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer.

Boyle, E., Chung, K.-M., and Pass, R. (2014). On extractability (aka differing-inputs) obfuscation.

Braun, B., Feldman, A. J., Ren, Z., Setty, S., Blumberg, A. J., and Walfish, M. (2013). Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357. ACM.

Cai, Y., Karakostas, G., and Wassyng, A. (2016). Secure and trusted white-box verification. *arXiv preprint arXiv:1605.03932*.

Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., and Waters, B. (2013). Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 40–49. IEEE.

Gentry, C. *et al.* (2009). Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178.

Gentry, C. and Halevi, S. (2011). Implementing gentrys fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*, pages 129–148. Springer.

Goldreich, O. (2004). *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA.

Goldreich, O. (2006). *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA.

Goldreich, O., Micali, S., and Wigderson, A. (1987). How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM.

Goldwasser, S. and Rothblum, G. N. (2007). On best-possible obfuscation. In *Proceedings of the 4th Conference on Theory of Cryptography*, TCC'07, pages 194–213, Berlin, Heidelberg. Springer-Verlag.

Goldwasser, S., Kalai, Y., Popa, R. A., Vaikuntanathan, V., and Zeldovich, N. (2013). Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564. ACM.

Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. (2001). A practical tutorial on modified condition/decision coverage.

Huang, Y., Evans, D., Katz, J., and Malka, L. (2011). Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201.

Janicki, R. and Wassyng, A. (2005). Tabular expressions and their relational semantics. *Fundamenta Informaticae*, **67**(4), 343–370.

Ji, Z. (2016). An implementation of secure verification using garbled circuits. Technical report, Dept. of Computing and Software, McMaster University.

Katz, J. and Lindell, Y. (2007). *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC.

Kaye, P., Laflamme, R., and Mosca, M. (2007). *An Introduction to Quantum Computing*. Oxford University Press, Inc., New York, NY, USA.

Kreuter, B., Shelat, A., and Shen, C.-H. (2012). Billion-gate secure computation with malicious adversaries. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 285–300.

Lindell, Y. and Pinkas, B. (2007). An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology-EUROCRYPT 2007*, pages 52–78. Springer.

Lindell, Y. and Pinkas, B. (2009). A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, **22**(2), 161–188.

Malkhi, D., Nisan, N., Pinkas, B., Sella, Y., *et al.* (2004). Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA.

Mohassel, P. and Riva, B. (2013). Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *Advances in Cryptology–CRYPTO 2013*, pages 36–53. Springer.

Mood, B., Gupta, D., Butler, K., and Feigenbaum, J. (2014). Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 582–596, New York, NY, USA. ACM.

Naor, M. (1991). Bit commitment using pseudorandomness. *Journal of Cryptology*, **4**(2), 151–158.

Naor, M. and Pinkas, B. (2001). Efficient oblivious transfer protocols. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457. Society for Industrial and Applied Mathematics.

Naor, M. and Pinkas, B. (2005). Computationally secure oblivious transfer. *Journal of Cryptology*, **18**(1), 1–35.

Nielsen, J. B. and Orlandi, C. (2009). Lego for two-party secure computation. In *Theory of Cryptography*, pages 368–386. Springer.

Parno, B., Howell, J., Gentry, C., and Raykova, M. (2016). Pinocchio: Nearly practical verifiable computation. *Commun. ACM*, **59**(2), 103–112.

Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. (2009). Secure two-party computation is practical. In *Advances in Cryptology–ASIACRYPT 2009*, pages 250–267. Springer.

Setty, S., Braun, B., Vu, V., Blumberg, A. J., Parno, B., and Walfish, M. (2013). Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 71–84. ACM.

Shen, C.-h. *et al.* (2011). Two-output secure computation with malicious adversaries. In *Advances in Cryptology–EUROCRYPT 2011*, pages 386–405. Springer.

Snyder, P. (2014). Yao's garbled circuits: Recent directions and implementations. Literature review, Dept. of Computer Science, University of Illinois at Chicago.

Vollmer, H. (1999). *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Walfish, M. and Blumberg, A. J. (2015). Verifying computations without reexecuting them. *Communications of the ACM*, **58**(2), 74–84.

Wang, W., Hu, Y., Chen, L., Huang, X., and Sunar, B. (2015). Exploring the feasibility of fully homomorphic encryption. *Computers, IEEE Transactions on*, **64**(3), 698–706.

Wassyng, A. and Janicki, R. (2003). Tabular expressions in software engineering. In *Proceedings of ICSSEA*, volume 3, pages 1–46.

Wee, H. (2005). On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM.

Yao, A. (1986). How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE.

Yao, A. C. (1982). Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE.