Secure and Trusted Verification

SECURE AND TRUSTED VERIFICATION

BY

YIXIAN CAI, 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 (2015)	McMaster University
(Computig & Software)	Hamilton, Ontario, Canada

TITLE:	Secure and Trusted Verification
AUTHOR:	Yixian Cai B.Eng., (Information Security Engineering) Shanghai Jiao Tong University, Shanghai, China
SUPERVISOR:	Dr. Karakostas, Dr. Wassyng
NUMBER OF PAGES:	x, 120

This thesis is dedicated to my parents.

Abstract

In our setting, verification is a process that checks whether a device's program (implementation) has been produced according to its corresponding requirements specification. Ideally a client builds the requirements specification of a program and asks a developer to produce the actual program according to the requirements specification it provides. After the program is built, a verifier is asked to verify the program. However, nowadays verification methods usually require good knowledge of the program to be verified and thus sensitive information about the program itself can be easily leaked during the process.

In this thesis, we come up with the notion of secure and trusted verification which allows the developer to hide non-disclosed information about the program from the verifier during the verification process and a third party to check the correctness of the verification result. Moreover, we formally study the mutual trust between the verifier and the developer and define the above notion in the context of both an honest and a malicious developer.

Besides, we implement the notion above both in the setting of an honest and a malicious developer using cryptographic primitives and tabular expressions. Our construction allows the developer to hide the modules of a program and the verifier to do some-what white box verification. The security properties of the implementation are also formally discussed and strong security results are proved to be achieved.

Acknowledgements

I would like to express my deepest appreciation to my supervisors, Dr. Karakostas and Dr. Wassyng. Without them offering me the opportunity, I would never have been able to be here and do research in this area. They really helped me a lot with my research and without them this thesis would not be possible. They are really great supervisors. They inspired me a lot and I learned a lot from them.

Moreover, I would like to thank the authors of the papers I read and cited in the thesis. Especially I need to thank Dr. Goldwasser and the other co-authors of the paper "Reusable Garbled Circuits and Succinct Functional Encryption". From their papers I learned a lot about cryptography and computational theory.

I also need to thank my friends who spiritually supported me during these two years when I felt lonely and homesick. Without them, these two years will not be as colourful as it is.

Finally, I thank my parents for supporting me to go abroad and coming to see me last summer. Their support is really important to me.

Notation and abbreviations

FHE, fully homomorphic encryption

hpk, the public key of FHE

hsk, the secret key of FHE

SE, private key encryption

sk, private key of SE

G, a normal table graph

TS, the set of the tables in G

PT_i, the ith table in TS

G', an encrypted table graph of G

PT_i', the encrypted table of PT_i

U, a universal circuit

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Verification is a process that checks whether a program has been produced according to its corresponding requirements specification (the program can either be a software program or a hardware device's design documentation). Nowadays verification methods usually require good knowledge of the program to be tested and thus sensitive information about the program itself can easily leak during the process. For instance, in white box testing the code of the program to be tested is required, which leaks everything about the program. Even in black box testing where the verifier has the program only in the form of an executable, partial information about the program itself can still leak. If the developer of the program is the one who provides the program's requirements specification and also the one who verifies whether the program behaves as expected, then it does not need to hide the content of the program. However, in most practical cases, the developer (DV) of the program, the verifier (V), and the client (CL) who provides the program requirements specification are not the same entity. The common scenario can be described as follows. CL builds the requirements specification of a program and asks another party (DV) to produce the actual program according to the requirements specification it provides. After the program is built, CL asks a third party (V) to verify the program against its requirements specification, or verifies it himself. The reasons that DV can not play the role of V are manifold. Firstly, CL may not trust DV and needs to verify whether DV has built the program correctly. Secondly, introducing a third party to verify the program may reduce the errors in the verification itself. In the above scenarios, if DV just gives the program code unprotected to V, V knows everything about the program easily. Hence if V is malicious, it can steal the intellectual property of DV related to this program. We need to point out here that protecting information of the program from leaking to the verifier is vital in many real life applications, despite the fact that nowadays open-source software is becoming more and more popular (and perhaps will become the mainstream one day). A typical example would be as follows: Car manufacturer A wants to buy car manufacturer B's automotive navigation system (which is B's intellectual property). Before A buys the system, A wants to verify whether the system is built according to its requirements specification. In such a scenario, if B gives the system to A or any third-party verifier without any special protection method, B risks leaking the system to A which may cost millions of dollars of economic loss.

On the other hand, V needs to know the program in detail in order to correctly verify the program. Therefore, if DV needs certain pieces of information about the program to be non-disclosed (which we call "non-disclosed information") during the verification of V, then the common scenario above is not applicable. Non-disclosed information in real life application can be the code of the program, or a particular module of the program.

Moreover, suppose there is a class of verification schemes that solves the above problem. Then these verification schemes still may lead to other problems regarding mutual trust between V and DV. Since these schemes prevent the non-disclosed information of the program from leaking to V, DV may probably somehow use this property to deceive V into believing that it does the verification correctly (even if the program will not pass V's verification in white-box verification). When DV is honest, this is not a problem. However, if DV is malicious, these verification schemes need to provide a mechanism to allow V to check whether it actually does the verification correctly (and if DV is honest).

Speaking of malicious activities, V can also be malicious. One scenario is that V does not verify correctly by making mistakes in the evaluation of the program (on purpose or unintentionally). Another scenario is that V attacks the verification scheme to figure out non-disclosed information about the program that DV hides. Hence verification schemes also need to take into consideration potential malicious activities on V's side.

Additionally, suppose the client asks V to verify DV's program and there is a way to ensure that DV and V are both honest. However, how can CL know whether V and DV collude during the verification to deceive him? In other words, CL also needs to be able to check the correctness of the verification. Broadly speaking, any third party should be able to check the correctness of the verification.

In conclusion, a *secure and trusted verification scheme* should not only protect the non-disclosed information of the program from leaking to V while allowing V

to correctly conduct the verification, but also provide a method for V and DV to trust each other (if both parties are honest) or help them discover the other party's malicious activities (if at least one party is malicious). Besides, the verification scheme should be publicly checkable, in order to provide a way to allow any third party to check the correctness of the verification process.

1.2 Previous Work

The most typical type of information that a developer wants to be non-disclosed is the content (code) of the program. This is also the focus of our work. In terms of protecting the content of a program , there is plenty of work that has already been done. One important concept is *obfuscation*. An obfuscated program is supposed to have the same functionality as the original program, but it reveals no information about the original program, other than what can be figured out by having black-box access to the original program. In this area, there are already plenty of heuristic obfuscation algorithms. General methods about how the code of a program should be obfuscated are also widely studied, e.g., in (Collberg *et al.*, 1997) and (Wang *et al.*, 2000). All these methods provide important guidance on achieving good obfuscation. However, according to (Barak *et al.*, 2001), an obfuscation algorithm that strictly satisfies the definition of obfuscation does not exist. Hence, all obfuscation algorithms can only achieve obfuscation to the extent of making obfuscated programs hard to reverse-engineer, while non black-box information about the original program cannot be guaranteed to be completely secret.

Though ideal obfuscation is impossible to achieve, there are still some other cryptographic primitives that can be used to protect the content of a program. One is *point function obfuscation* (Lynn *et al.*, 2004). This paper shows that obfuscation of a point function is totally possible, where a point function is a function that outputs 1 for one specific value as input and outputs 0 otherwise. This obfuscation scheme is useful when it comes to protecting sensitive data in a program. But it does not support obfuscation of arbitrary functions.

Another relevant cryptographic primitive is Yao's *garbled circuit* (Yao, 1982). It requires encoding of an arbitrary circuit C and its input x. Evaluation of the encoded (garbled) circuit will generate $C(x)$ without leaking any information about C or x other than $C(x)$. Yao's garbled circuit scheme achieves nearly what obfuscation does in terms of one-time usage, and is the foundation of works like one-time program (Goldwasser *et al.*, 2008). Its problem is that the garbled circuit can only be run once. Many-time evaluation on the same garbled circuit would compromise the security of the garbled circuit. Recently, Goldwasser et al. (Goldwasser *et al.*, 2013) proposed a new garbled circuit scheme called *reusable garbled circuit*, which allows many-time usage on the same garbled circuit and fully satisfies the security property of Yao's garbled circuit. In the same paper, Goldwasser et al. proposed the notion of *token-based obfuscation* which is based on reusable garbled circuits. Token-based obfuscation allows a user to run the token-based obfuscated program himself. He can also freely choose any input as it likes. The difference between this obfuscation and the obfuscation definition in (Barak *et al.*, 2001) is that this obfuscation needs the user to ask the developer who obfuscates the program to encode the input before the user uses it. Token-based obfuscation guarantees that only black-box information about the original program can be figured out from the obfuscated program. Thus it is an ideal tool to protect the content of a program. The only problem about this scheme is that when a program is token-based obfuscated, it becomes a black-box to the user (due to the inherent nature of obfuscation), and thus makes white-box verification hard to implement. Both token-based obfuscation and reusable garbled circuit include the use of *fully homomorphic encryption* (FHE), which is a very powerful encryption concept that allows computation over encrypted data. Gentry came up with the first FHE scheme in 2009 (Gentry, 2009). FHE is also used in our construction.

On the other hand, plenty of work has been done in the field of verifiable computing which is also relevant to our work. Verifiable computing allows the prover to generate verifiable results of the computation of a function. A verifier (this is not the verifier in our setting) can then verify whether the computation is done correctly according to the verifiable results. A naive solution is to ask the verifier to repeat the computation of the function. However, in many occasions this solution is both inefficient and incorrect. Recently, works like (Cormode *et al.*, 2012), (Thaler *et al.*, 2012), (Vu *et al.*, 2013), (Setty *et al.*, 2012a), (Setty *et al.*, 2012b), (Parno *et al.*, 2013), (Ben-Sasson *et al.*, 2013) which are based on the PCP theorem (Arora and Safra, 1998) implemented systems that allow the verifier to verify the computation result without reexecuting the function. The difference between verifiable computing and our work is that one important goal of our work is to hide the content of a program, while for verifiable computing, the program that needs to be computed is not hidden. So it can not be directly applied to our work. However, verifiable computing is relevant to our work and has the potential to be applied to the implementation of secure and trusted verification (see Section 5.2.1).

1.3 Our contribution

Our contribution is threefold. Firstly, we give definitions regarding the notion of secure and trusted verification (STV). Our definitions generalize the problem of protecting non-disclosed information of a program from leaking to the verifier while allowing him to correctly verify the program. Furthermore, our definitions describe the above problem both in the context of an honest and a malicious developer. Moreover we add the concept of third-party verification to our verification definitions. In our definitions any third party should also be able to check whether the verifier does the verification correctly and whether the developer is honest.

Secondly, we come up with an implementation of STV. In our implementation, a program to be verified is regarded as a directed graph of the modules of the program connected with each other. During the execution of the program, the modules will output information and receive other modules' information according to the graph. The non-disclosed information in our implementation is the code of the modules and the outputs of the modules that do not belong to the output of the program (the output of the program is considered as public knowledge). Our implementation guarantees that the non-disclosed information will not leak to the verifier even if the verifier is malicious. Moreover, our implementation allows the verifier to verify the program correctly regardless of whether the developer is honest or malicious. Besides, our implementation is third-party checkable, which means that any third party can check if the verifier does the verification correctly and if the developer is honest. Our construction is based on the fully homomorphic encryption (FHE) mentioned in section 1.2 and the bit commitment protocol (Naor, 1991). By using FHE we solve the problem of protecting non-disclosed information and by using bit commitment protocol we prevent the developer from being malicious.

Thirdly, we introduce the use of tabular expressions in our implementation of STV. We propose the idea of using tabular expressions in (Wassyng and Janicki, 2003) to represent the program specifics. The final program given to the verifier will be in the form of a graph of encrypted tabular expressions, where the tabular expressions describe the modules of the program. By adopting this idea, the branch structure and data flow of the program will be exposed to the verifier, while the program specifics and the intermediate inputs and outputs of the tabular expressions are protected. Hence at least some extent of white-box verification of the program is feasible.

Below we list the important assumptions we make. Our implementations' correctness and security properties in Chapter 3 and 4 are based on these assumptions. These assumptions will be explained later when used.

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

Assumption 2. *Given the table graph G of a program, the input domain and output range of G are the same as the input domain and output range of the corresponding program's requirements specification F_{spec}.*

Assumption 3. *For any path of a table graph, there is at least one external input to the table graph such that evaluating the table graph with this external input will include evaluating the path.*

1.4 Organization

In Chapter 2 we give some basics about FHE, tabular expressions and introduce the definitions of some cryptographic primitives which are the foundation of our construction. The format of tabular expressions of a program will also be discussed in detail.

In Chapter 3 we give the definition of STV in the setting of an honest developer. We also explain our construction of this verification scheme in this chapter and give a proof to show the construction satisfies the definition. This chapter mainly discusses the problem of how to hide the content of a program.

In Chapter 4 we give the definition of STV in the setting of a malicious developer. We also explain our construction of this verification scheme in this chapter and give a proof to show the construction satisfies the definition. This chapter mainly discusses how to detect potential malicious activities of the developer.

In Chapter 5 we discuss some problems open for future study.

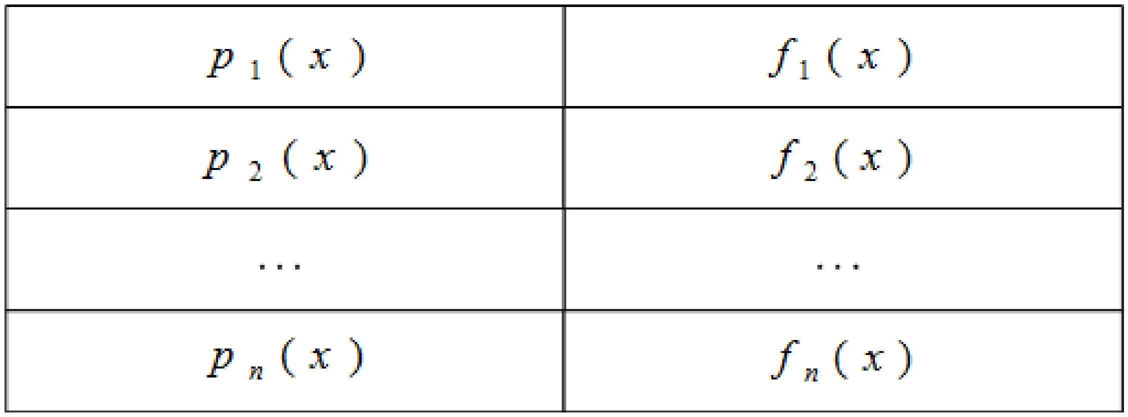
Chapter 2

Preliminaries

2.1 Tables

Tables (tabular expressions) are used for the documentation of software programs. The concept was first introduced by David Parnas in the 1970s (Alspaugh *et al.*, 1992). Since then the semantics and syntax of tables have been widely discussed and various types of tables were developed. The tables we use are from (Wassyng and Janicki, 2003), which are already used in some critical software development and have relative simple syntax and semantics. Figure 2.1 is an example of such a table.

In Figure 2.1 we can see that a table is divided into two columns: the left hand side (lhs) *Condition* column and the right hand side (rhs) *Result* column. The conditions in the lhs column are predicates while the contents in the rhs column are functions (constant values are regarded as zero-arity functions). If a predicate $p_i(x)$ is satisfied by an input x, then $f_i(x)$ will be the output of the table. Besides, the conditions have disjointness and completeness properties: $p_1(x) \vee p_2(x) \vee ... \vee p_n(x) = True$; $p_i(x) \wedge p_j(x) = FALSE, \forall i, j \in [n]$. Hence a table can be used to represent a module



$p_1(x)$	$f_1(x)$
$p_2(x)$	$f_2(x)$
...	...
$p_n(x)$	$f_n(x)$

Figure 2.1: A table with input x

of a program and with different tables connecting to each other we can document a complete program in the form of tables.

2.2 Table graphs

2.2.1 Initial table graph

In our construction in Chapter 3 and 4 we assume that we have the program in the form of tables connecting to each other where the tables are the modules of the program. We call the resulting graph with the vertices being the tables as a *table graph*. The details of how to transform a program to a table graph is beyond the scope of this thesis.

First we give our notations of concepts related to a table. A table with n rows is denoted as $T = (r_1, r_2, ...r_n)$, where for any $j \in \{1, ..., n\}$, $r_j = (p_j, f_j)$ is the jth row of T, with p_j being the lhs predicate of r_j and f_j being the rhs function of r_j.

Suppose x is the input to T. Then $T(x)$, the output of T, is defined as follows: if for a row $r_j = (p_j, f_j)$ of T, $p_j(x) = 1$ (the predicate p_j is satisfied by the input x), then $T(x) = f_j(x)$.

A table graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed acyclic graph. (According to Assumption 1, we only consider the case where a table graph has no cycles or self loops.) There are two special nodes S and R in $\mathcal{V}$. S represents the external inputs to the program and R represents the outputs of the program. $\mathcal{RS} = \mathcal{V} - \{R, S\}$ is the set of rows and every vertex in $\mathcal{RS}$ is a row of a table. We denote $\mathcal{RS} = \{r_1, ..., r_l\}$, assuming there are l rows in total. The reason that a vertex in a table graph is not a table but a row can be illustrated with the example in Figure 2.2. In Figure 2.2 we can see that the first three rows of PT_1 point to rows of table PT_2. But the fourth row of PT_1 points to the output vertex R. This situation cannot be represented by using tables as vertices. In this example vertices are the rows of the tables. Some rows in the figure are put together to show that they come from the same table. Essentially in a table graph the rows of a table do not need to be put together like in this figure.

For any edge $e = (u_i, u_j)$, $e \in \mathcal{E}$, u_i, $u_j \in \mathcal{V}$ is defined as follows:

(1) if $u_i = r_i$ and $u_j = r_j$, $r_i, r_j \in \mathcal{RS}$, then e indicates that the output of r_i belongs to the inputs of r_j.

(2) If $u_i = S$ and $u_j = T_j$, $r_j \in \mathcal{RS}$, then e indicates that the input of r_j has external inputs.

(3) If $u_i = r_i$ and $u_j = R$, $r_i \in \mathcal{RS}$, then e indicates that r_i's output is an external output.

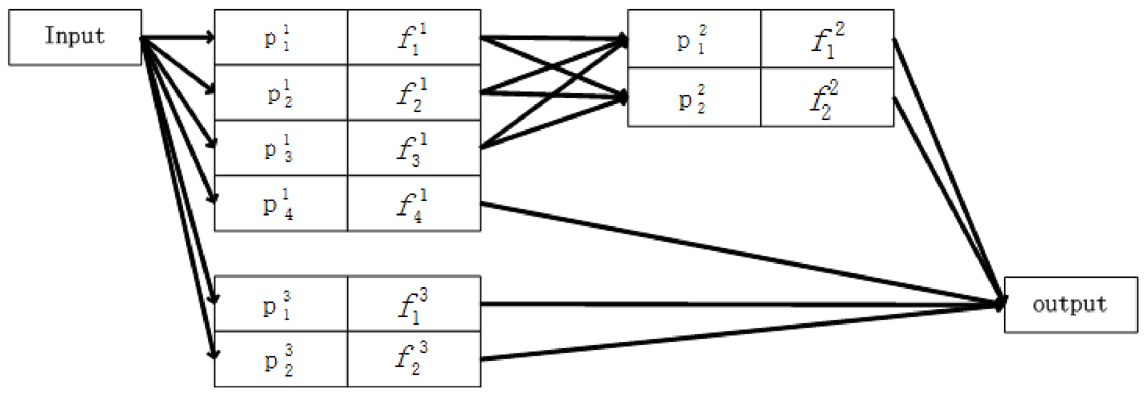


Figure 2.2: An initial table graph $\mathcal{G}$

2.2.2 The table graph in our construction

For our construction in Chapter 3 and 4 to work, we need a transformation of the table graph introduced above. The transformation is to split every table in the initial table graph into smaller tables, where each smaller table is a row of the original table. This would give us a table graph of better granularity while preserving original table graph functionality. To transform a row in the initial table graph to an independent table, we put "True" on the lhs of the table so that the rhs function will always be evaluated. The rhs function of the new table will be a piecewise function that has the same functionality as the rhs function of the row when the predicate on the lhs of the row is satisfied by the input. The rhs function of the new table will output a special symbol $\perp$ if the predicate on lhs of the row is not satisfied. Besides, it will output a special symbol $\top$ if the predicate on lhs of the row is satisfied.

We denote the table graph after the transformation as $G = (V, E)$. The definition of G is given below:

Definition 1. *For an initial table graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the corresponding table graph*

$G = (V, E)$ after the transformation is as follows:

(1) V has two special nodes S, R which are the same as the S, R in $\mathcal{V}$. We denote TS to be the table set of G. $TS = V - \{S, R\}$.

(2) Suppose the input x of any row $r_i \in \mathcal{RS}$ is $(x_1, x_2, ..., x_s)$, where x_i is one of the inputs that corresponds to an incoming edge of r_i. Then we define F_i as follows: F_i's input $w^i = (w^i_1, ..., w^i_s)$, where for any w^i_j, $w^i_j = (\top, x_j)$.

$F_i((w^i_1, ..., w^i_s)) = (\top, r_i(x))$, if $p_i(x) = 1$; $F_i((w^i_1, ..., w^i_s)) = (\bot, \bot)$, if $p_i(x) = 0$. If any $w^i_j = (\bot, \bot)$, $F_i((w^i_1, ..., w^i_s)) = (\bot, \bot)$. ($\top$ and $\bot$ are special symbols to indicate whether the predicate p^i_j is satisfied by x. They are not boolean values "True" and "False".) Moreover, we construct $PT_i \in TS$ such that $PT_i = (True, F_i)$. Thus $TS = \{PT_1, PT_2, ..., PT_n\}$.

(3) We require that for any table $PT_i \in TS$, if the first parameter of its output is $\bot$, then this output should not be used as an input to the subsequent tables which PT_i has an edge pointing to.

(4) For any edge $e = (u_i, u_j)$, $e \in \mathcal{E}$, u_i, $u_j \in \mathcal{V}$

1 if $u_i = r_i$ and $u_j = r_j$, $r_i, r_j \in \mathcal{RS}$, then for the corresponding two tables $PT_i, PT_j \in TS$ generated from r_i and r_j according to step (2), $(PT_i, PT_j) \in E$.

2 if $u_i = S$ and $u_j = r_j$, $r_j \in \mathcal{RS}$, then for the corresponding table $PT_j \in TS$ generated from r_j according to step (2), $(S, PT_j) \in E$.

3 if $u_i = r_i$ and $u_j = R$, $r_i \in \mathcal{RS}$, then for the corresponding table $PT_i \in TS$ generated from r_i according to step (2), $(PT_i, R) \in E$.

Figure 2.3 is an example which is the transformation result of Figure 2.2. In this example every vertex is a table ($PT_i = (True, F_i)$, $i \in \{1, ..., 8\}$). In Definition 1(2), $F_i(x)$ always outputs two values, either $(\top, r_i(x))$ or $(\bot, \bot)$. By that we mean each of the two values take half of the total output's length. The reason of adding $\top, \bot$ to F_i 's output will be explained in Chapter 4.

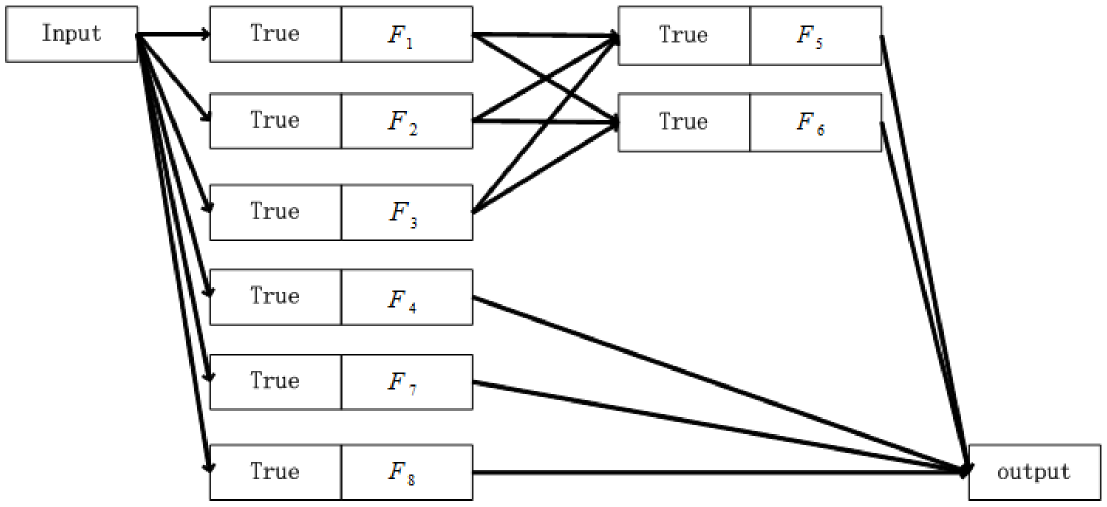


Figure 2.3: A table graph G

From now on when we mention a table graph, we refer to the table graph introduced in Definition 1. For a table graph G, we define an *external input* to be an input that comes directly from S and an *external output* to be an output that goes directly to R. An *intermediate input* is an input that comes from the output of another table and an *intermediate output* is an output that is an input to another table.

2.2.3 The structure graph of a table graph

The vertices of a table graph are not meaningless nodes but tables which represent details of a program. In our setting we can only know the relations between the vertices of a table graph, and not the vertices' content because the developer does not want to reveal it. Hence we need a new type of graph to describe the relations between the vertices of a table graph G, which we call the *structure graph* G_{struc} of the table graph G. Figure 2.4 is a structure graph of the table graph in Figure 2.3 where the connections between vertexes 1-8 show the connections between $PT_1, ..., PT_8$ in Figure 2.3.

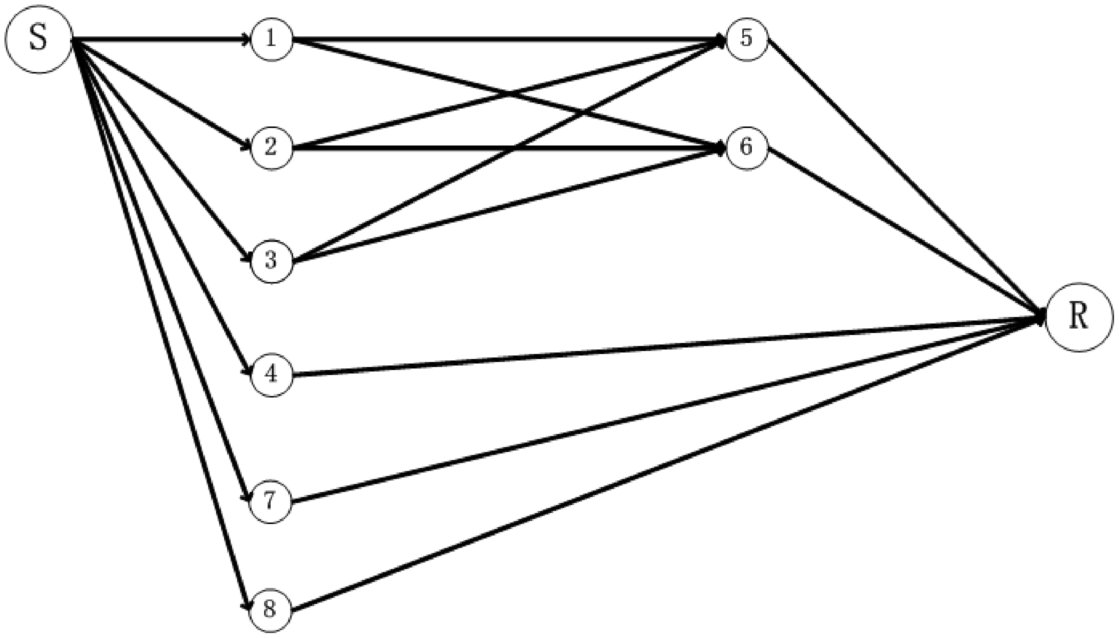


Figure 2.4: The structure graph G_{struc} of table graph G

Definition 2. *A structure graph* $G_{struc} = (V_{struc}, E_{struc})$ *of the table graph* $G = (V, E)$ *is defined as follows:*

(1) V_{struc} has two special nodes S, R which are the same as S, R in V.

(2) For any vertex $PT_i \in TS = V - \{S, R\}$, there is a corresponding vertex $u_i \in V_{struc}$ which is just an empty node.

(3) For any edge $e = (v_i, v_j)$, $e \in E$, v_i, $v_j \in V$

1) If $v_i = PT_i$ and $v_j = PT_j$, then for the two corresponding vertices $u_i, u_j \in V_{struc}$, $(u_i, u_j) \in E_{struc}$.

2) If $v_i = S$ and $v_j = PT_j$, then for the corresponding two vertices $S, u_j \in V_{struc}$, $(S, u_j) \in E_{struc}$.

3) If $v_i = PT_i$ and $v_j = R$, then for the two corresponding vertices $u_i, R \in V_{struc}$, $(u_i, R) \in E_{struc}$.

Now that we have defined table graph G and its structure graph G_{struc}, we are going to give a notation related to the table graph. First we denote the operation "[]" as follows:

$$G = [TS, G_{struc}]$$

Operation "[]" replaces the empty vertexes in G_{struc} with the corresponding tables in TS and thus reconstructs G from TS and G_{struc}. This operation shows that G consists of two non-overlapping parts which are TS and G_{struc}.

2.3 The verifier and the verification algorithm

We denote the function F_{spec} to represent a program's requirements specification, which documents the intended behaviours of the program. F_{spec} has its input domain, which we denote as D_{spec}, and its output range, which we denote as R_{spec}. We denote $X = \{x_1, ..., x_t\}$ as F_{spec}'s set of input variables. Consequently, D_{spec} is a multi-dimensional set such that $D_{spec} = \{D_1, ..., D_t\}$, where $D_i \in D_{spec}$ is the domain of input x_i. Similarly, we denote by $Y = \{y_1, ..., y_s\}$ F_{spec}'s set of output variables, and by $R_{spec} = \{R_1, ..., R_s\}$, where R_i is the range of y_i, the set of ranges of outputs Y.

Besides, the corresponding program's table graph G also has its input domain D and output range R. Because the program is an implementation of the requirements specification F_{spec}, for simplicity, we assume $D = D_{spec}$, $R = R_{spec}$ (see Assumption 2). (Although in reality, the two pairs (D, R) and (D_{spec}, R_{spec}) may not be the same. The developer may for some reason produce a program that has a different input domain other than the input domain of F_{spec}. Nevertheless, in this case there must be a predetermined mapping provided by the developer to convert (D_{spec}, R_{spec}) to (D, R) and same vice versa.)

Below we define the *level* of a table in G that will be used later in the thesis.

Definition 3. *For a table graph G, the level of a table is defined inductively as follows:*

(1) An external input to a table can be regarded as the output of a virtual level 0 table.

(2) A level k ($k > 0$) table is a table such that it has at least one incoming edge that is from a level $k - 1$ table and no incoming edge comes from a table whose level is larger than level k -1.

The input to a level k ($k > 0$) table PT_i in G is $x^i = (x^i_1, ..., x^i_s)$, where $x^i_j \in x^i$ denotes an input variable which corresponds to an incoming edge from a table before level k. The output variable of PT_i is $y_i = PT_i(x^i)(= F_i(x^i))$. For any $x^i_j \in x^i$, if no value is assigned to x^i_j, x^i_j's value is *null*. The *evaluation* of a single table PT_i with its input x^i is the computation of $y_i = PT_i(x^i)(= F_i(x^i))$. If PT_i is not evaluated, then we regard the output of PT_i as *null*. (A level k ($k > 0$) table also can be regarded as a table such that among all its paths starting from a level 0 table, the longest path is of length k.)

Given an external input X to the graph G, the *evaluation* of G, which is denoted as $G(X)$, is described below.

Definition 4. *The evaluation of a table graph G with an external input X, is defined below:*

(1) The evaluation of G must evaluate the tables in G in the order of a topological order $TSO = (PT_{s_1}, ..., PT_{s_n})$ of G such that PT_{s_1} is a source and PT_{s_n} is a sink.

(2) Suppose $PT_{i_1}, ..., PT_{i_s}$ are the tables that have external outputs. Then their outputs belong to the external output of G. Namely, after finishing evaluation of every table in the order of TSO, $G(X) = \{y_{i_1}, ..., y_{i_s}\}$, where $\{y_{i_1}, ..., y_{i_s}\}$ are the outputs of $PT_{i_1}, ..., PT_{i_s}$.

Definition 5 (Consistency). *When evaluating a level k table PT_i with its input x^i, for any $x^i_j \in x^i$, x^i_j's value can only be the output of another level o ($o < k$) table where there is an outgoing edge from that table to PT_i. If x^i_j is $(\perp, \perp)$ or if x^i_j is* null, *then we skip evaluating PT_i and regard the output of PT_i as* null.

Remark 1. *Definition 5 follows Definition 4. Besides, our definitions and implementations of secure and trusted and verification do not require the verifier to be consistent.*

We split the verification process into two parts. One part is to analyse the table graph G with a verification method VGA (Verification test case Generation Algorithm), and to generate a set of external inputs. The second part is to use the external inputs to evaluate the table graph, and compare the outputs of G to the outputs of F_{spec}. If they all match, then the table graph G behaves as expected with the external inputs, which means that G passes the verification by VGA.

In our setting VGA is an algorithm that takes G_{struc}, F_{spec} as input and outputs information that guides the verification such as a set of external inputs to the table graph G or certain paths of G. Here we point out that the output of VGA is not necessarily the external inputs. The reason is related to our construction in chapter 3 and 4. In our construction we give as much freedom to the verifier as we can. (By freedom we mean the choice of VGA.) Some VGA may not be able to directly generate external inputs by analysing a encrypted version of G (after applying our verification scheme to G.) But these VGA can generate instead information which provides guidance as how to generate the external inputs, such as pointing out the paths of the graph which these VGA want to evaluate. After processing this information (and interact with the developer using a protocol of the verification scheme), the verifier can convert it into a set of external inputs.

On the other hand, the developer may also want the verifier to verify the behaviour of the table graph G (or a subgraph of G which can be a module of a program) with particular external input values and expected outputs given by the developer, which we denote as the critical point set CP.

Definition 6. *For any $CP_i \in CP$, CP_i is a pair (X, Y), where X is an external input to a subgraph of G which we denote as G_X and Y is the external output of G_X. G_X can be determined by X as the following:*

(1) Suppose $X = (x_1, ..., x_p)$. Then for any table $PT_{t_i} \in G$, if the input to PT_{t_i} consists of only external inputs $x_{i_1}, ..., x_{i_q} \in X$, put PT_{t_i} into a set S.

(2) For this subgraph, we define the tables in S as positive *tables. The rest tables in G are initially labelled as* negative *tables. Then we relabel the* negative *tables of G in a topological order to find the vertices of the subgraph G_X. We set a* negative *table's label to* positive *if all of its incoming edges are from* positive *tables. After trying to relabel all the* negative *tables, G_X consists of all the* positive *tables in G.*

Definition 7 (Verifier and the verification algorithm)**.** *A verifier V is a polynomial-time algorithm such that*

$$V(1^K, G, F_{spec}, CP, VGA) = \begin{cases} 0 & , \exists X \in EI\text{: } G(X) \neq F_{spec}(X) \text{ or} \\ & \exists CP_i = (X, Y) \in CP\text{: } G_X(X) \neq Y, \\ 1 & , otherwise, \end{cases}$$

where EI is generated by V, $EI \subseteq D$. (If G is unencrypted, V does not need to interact with the verifier.)

2.4 Notations and definitions

Below are some basic widely-known cryptography notations and definitions that are referred to throughout the rest of the thesis. We denote $negl(K)$ to be the *negligible function* with an input parameter $K \in \mathbb{N}$ such that for all $c > 0$, there exists N such that for all $K > N$, $negl(K) < K^{-c}$. The notation p.p.t. is the abbreviation for *probabilistic polynomial-time turing machine*. We say that two ensembles $\{X_K\}_{K \in \mathbb{N}}$ and $\{Y_K\}_{K \in \mathbb{N}}$, where X_K, Y_K are probability distributions over $\{0, 1\}^{l(K)}$ for a polynomial

$l(\cdot)$, are *computationally indistinguishable* if for every p.p.t. algorithm D,

$$|Pr[D(X_K, 1^K) = 1] - Pr[D(Y_K, 1^K) = 1]| \leq negl(K).$$

If $\{X_K\}_{K\in\mathbb{N}}$ and $\{Y_K\}_{K\in\mathbb{N}}$ are computationally indistinguishable, we also denote them as $\{X_K\}_{K\in\mathbb{N}} \approx \{Y_K\}_{K\in\mathbb{N}}$ (We use the definitions of negligible function and computational indistinguishability in (Goldwasser *et al.*, 2013) with some adaptations). Below we give the definition of a deterministic private key encryption (symmetric key encryption) scheme from (Katz and Lindell, 2014) with some adaptations.

Definition 8 (Private key encryption scheme). *A private key enryption scheme SE is a tuple of three polynomial-time algorithms $(SE.KeyGen, SE.Enc, SE.Dec)$ as follows:*

(1) The key generation algorithm $SE.KeyGen$ is a probabilistic algorithm that takes the security parameter K as input and outputs a key sk from the key space $\mathcal{K} = \{0,1\}^K$.

(2) The encryption algorithm $SE.Enc$ is a deterministic algorithm that takes a key sk and a plaintext M from the message space $\mathcal{M} = \{0,1\}^{m(K)}$, $m(K) > K$ as input and outputs a ciphertext C. Namely, $C = SE.Enc(sk, M)$.

(3) The decryption algorithm $SE.Dec$ is a deterministic algorithm that takes as input a key sk and a ciphertext C and outputs the corresponding plaintext M. Namely, $M = SE.Dec(sk, C)$.

Message indistinguishability requires that an adversary must not be able to distinguish two ciphertexts even if it chooses the plaintexts of these two pieces of ciphertexts. Below we give the definition:

Definition 9 (Single message indistinguishability)**.** *A private key encryption scheme $SE = (SE.KeyGen, SE.Enc, SE.Dec)$ is* single message indistinguishable *if for any security parameter K, for any two messages $M, M' \in \mathcal{M}$, and for any p.p.t. adversary A,*

$$|Pr[A(1^K, SE.Enc(k, M)) = 1] - Pr[A(1^K, SE.Enc(k, M')) = 1]| \leq negl(K)$$

where the probabilities are taken over $k \leftarrow SE.KeyGen(1^K)$ and the coin tosses of A.

The private key encryption scheme SE we use in our construction in Chapter 4 satisfies Definition 8 and Definition 9. An example of such a private key encryption would be a block cipher such as Data Encryption Standard (DES) (Standard, 1977).

2.5 Fully homomorphic encryption

In this section we introduce a powerful cryptographic primitive that we need in our construction of content-secure verification scheme in Chapter 3 and 4, the *fully homomorphic encryption scheme.*

The following definitions are from (Goldwasser *et al.*, 2013), and are based on (Vaikuntanathan, 2011) with some adaptations.

Definition 10 (Homomorphic encryption)**.** *A homomorphic (public-key) encryption scheme HE is a quadruple of polynomial time algorithms $(HE.KeyGen, HE.Enc, HE.Dec, HE.Eval)$:*

(1) $HE.KeyGen(1^K)$ is a probabilistic algorithm that takes as input the security parameter 1^K and outputs a public key hpk and a secret key hsk.

(2) $HE.Enc(hpk, x \in \{0,1\})$ is a probabilistic algorithm that takes as input the public key hpk and an input bit x and outputs a ciphertext ϕ.

(3) $HE.Dec(hsk, \phi)$ is a deterministic algorithm that takes as input the secret key hsk and a ciphertext ϕ and outputs a message $x' \in \{0,1\}$.

(4) $HE.Eval(hpk, C, \phi_1, ..., \phi_n)$ is a deterministic algorithm that takes as input the public key hpk, a circuit C that takes n bits as input and outputs one bit, as well as n ciphertexts $\phi_1, ..., \phi_n$. It outputs a ciphertext ϕ_C.

Compactness: *For all security parameters K, there exists a polynomial $p(\cdot)$ such that for all input sizes n, for all $x_1, ..., x_n$ and C, the output length of HE.Eval is at most $p(n)$ bits long.*

Definition 11. *Let $C = \{C_n\}_{n \in \mathbb{N}}$ be a class of boolean circuits, where C_n is a set of boolean circuits taking n bits as input. A scheme HE is C-homomorphic if for every polynomial n, sufficiently large K, $C \in C_n$, and for every input bit sequence $x_1, ..., x_n$, where n=n(K),*

$$Pr\begin{bmatrix} (hpk, hsk) \leftarrow HE.KeyGen(1^K); \\ \phi_i \leftarrow HE.Enc(hpk, x_i) \text{ for } i = 1, ..., n; \\ \phi \leftarrow HE.Eval(hpk, C, \phi_1, ..., \phi_n): \\ HE.Dec(hsk, \phi) \neq C(x_1, ..., x_n) \end{bmatrix} \leq negl(K) \tag{2.1}$$

where the probability is over the coin tosses of $HE.KeyGen$ and $HE.Enc$.

Definition 12. *A scheme HE is fully homomorphic if it is homomorphic for the class of all arithmetic circuits over GF(2).*

Definition 13. *A scheme HE is IND-CPA secure if for any p.p.t. adversary A,*

$$|Pr[(hpk, hsk) \leftarrow HE.KeyGen(1^K) : A(hpk, HE.Enc(hpk, 0)) = 1] -$$
$$Pr[(hpk, hsk) \leftarrow HE.HeyGen(1^K) : A(hpk, HE.Enc(hpk, 1)) = 1]| \leq negl(K).$$

The following notion and Definition 14 of multi-hop homomorphism are from (Gentry *et al.*, 2010). Multi-hop homomorphism is about using the output of one homomorphic evaluation as an input for another homomorphic evaluation. Not all homomorphic encryption schemes support this property inherently.

An ordered sequence of functions $\vec{f} = \{f_1, ..., f_t\}$ is *compatible* if the output length of f_j is the same as the input length of f_{j+1} for all j. The composed function $f_t(...f_2(f_1(\cdot))...)$ is denoted as $(f_t \circ ... \circ f_1)(x)$. An extended procedure $Eval^*$ is defined as follows: $Eval^*$ takes as input the public key PK, a compatible sequence $\vec{f} = \{f_1, ..., f_t\}$, and a ciphertext c_0. For $i = 1, 2, ..., t$, it sets $c_i \leftarrow Eval(PK, f_i, c_{i-1})$, outputting the last ciphertext c_t.

Definition 14. *Let $i = i(K)$ be a function of the security paramenter. A scheme $HE = (HE.KeyGen, HE.Enc, HE.Dec, HE.Eval)$ is an i-hop homomorphic encryption scheme if for every compatible sequence $\vec{f} = \{f_1, ..., f_t\}$ with $t \leq i$ functions, every input x to f_1, every (hpk, hsk) in the support of HE.KeyGen, and every c in the support of $HE.Enc(hpk, x)$,*

$$HE.Dec(hsk, Eval^*(hpk, \vec{f}, c)) = (f_t \circ ... \circ f_1)(x)$$

We say that HE is a multi-hop homomorphic encryption scheme if it is i-hop for any polynomial i.

From (Gentry *et al.*, 2010) (Vaikuntanathan, 2011) we know that multi-hop evaluation issues are trivialized for fully homomorphic encryption schemes. Also, we require the fully homomorphic encryption scheme FHE used in our construction in Chapter 3 and 4 to satisfy IND-CPA security in Definition 13.

Fully homomorphic encryption has been only a concept for a long time. But recently some concrete implementations of fully homomorphic encryption have been invented. An example is Gentry's first FHE implementation in the breakthrough paper (Gentry, 2009). In this thesis we do not specify what kind of FHE implementation we use. Any FHE scheme can be applied to our construction in Chapter 3 and 4.

For simplicity, we sometimes denote $FHE.Enc(hpk, x)$ as $FHE.Enc(x)$ when we do not care about the public key hpk. Moreover, when $x = x_1...x_m$ is a m bit string and $m > 0$, we use $FHE.Enc(x)$ to denote $FHE.Enc(x_1)...FHE.Enc(x_m)$. This applies to $FHE.Dec$ as well.

Similarly, for $FHE.Eval$ with a circuit C as its input such that C outputs m bits, sometimes we use $FHE.Eval(hpk, C, FHE.Enc(hpk, x))$ to denote $FHE.Eval(hpk, C_1, FHE.Enc(hpk, x))...FHE.Eval(hpk, C_m, FHE.Enc(hpk, x))$, where C_i is a circuit that outputs the ith bit of C's output.

Also we denote by $\lambda = \lambda(K)$ the ciphertext length of a one-bit FHE encryption.

2.6 Bit commitment protocols

Below we give the definition of the Commitment to Many Bits Protocol by Naor (Naor, 1991) with some adaptations.

Definition 15 (Commitment to Many Bits Protocol)**.** *A commitment to many bits protocol consists of two stages*

(1) The commit stage: Alice has a sequence of bits $D = b_1 b_2 ... b_m$ to which she wishes to commit to Bob. She and Bob exchanges messages. At the end of the stage Bob has some information that represents D which is denoted as Enc_D.

(2) The revealing stage: Bob knows D at the end of this stage.

The protocol must obey the following: For any probabilistic polynomial time Bobs, for all polynomials p and for large enough security parameter K

(1) For any two sequence $D = b_1, b_2, ..., b_m$ and $D' = b'_1, b'_2, ..., b'_m$ selected by Bob, following the commit stage Bob cannot guess whether D or D' was committed with probability greater than $1/2 + 1/p(K)$.

(2) Alice can reveal only one possible sequence of bits. If she tries to reveal a different sequence of bits, then she is caught with probability at least $1 - 1/p(K)$.

An example of a commitment to many bits protocol is the construction in (Naor, 1991). This protocol will be used in our construction of STV in Chapter 4 and is given below.

Let $C \subset \{0,1\}^q$ be a code of 2^m words such that the Hamming distance between any $c_1, c_2 \in C$ is at least $\epsilon \cdot q$. E is an efficiently computable function such that $E : \{0,1\}^m \to \{0,1\}^q$ for mapping words in $\{0,1\}^m$ to C. It is also required that $q \cdot log(2/(2-\epsilon)) \geq 3K$ and $q/m = c$, where c is a fixed constant.

G denotes a pseudo-random generator G:$\{0,1\}^K \to \{0,1\}^{l(K)}$, $l(K) > K$ such that for all p.p.t. adversary A ,

$$|Pr[A(y) = 1] - Pr[A(G(s)) = 1] < 1/p(K),$$

where the probabilities are taken over $y \in \{0,1\}^{l(K)}$ and seed $s \in \{0,1\}^K$ chosen uniformly at random. $G_k(s)$ denotes the first k bits of the pseudo-random sequence on seed $s \in \{0,1\}^K$ and $B_i(s)$ denotes the ith bit of the pseudo-random sequence on seed s. For a vector $\vec{R} = (r_1, r_2, ..., r_{2q})$ with $r_i \in \{0,1\}$ and q indices i such that $r_i = 1$, $G_{\vec{R}}(s)$ denotes the vector $\vec{A} = (a_1, a_2, ..., a_q)$ where $a_i = B_{j(i)}(s)$ and $j(i)$ is the index of the ith 1 in $\vec{R}$. If $e_1, e_2 \in \{0,1\}^q$, then $e_1 \oplus e_2$ denotes the bitwise Xor of e_1 and e_2.

Suppose Alice commits to $b_1, b_2, ..., b_m$.

Commit Stage.

(1) Bob selects a random vector $\vec{R} = (r_1, r_2, ..., r_{2q})$ where $r_i \in \{0,1\}$ for $1 \leq i \leq 2q$ and exactly q of the r_i's are 1 and sends it to Alice.

(2) Alice computes $c = E(b_1, b_2, ..., b_m)$. Alice selects a seed $S \in \{0,1\}^n$ and sends to Bob Enc_D which is the following: Alice sends Bob $e = c \oplus G_{\vec{R}}(s)$ (the bitwise Xor of $G_{\vec{R}}(s)$ and c), and for each $1 \leq i \leq 2q$ such that $r_i = 0$ she sends $B_i(s)$.

Reveal Stage.

Alice sends s and $b_1, b_2, ..., b_m$. Bob verifies that for all $1 \leq i \leq 2q$ such that $r_i = 0$ Alice had sent the correct $B_i(s)$, computes $c = E(b_1, b_2, ..., b_m)$ and $G_{\vec{R}}(s)$ and verifies that $e = c \oplus G_{\vec{R}}(s)$.

We denote $\vec{R}$ as R, the bits $b_1, b_2, ..., b_m$ as D, the seed s as $Cert$ and give below the protocol of the above construction in Figure 2.5.

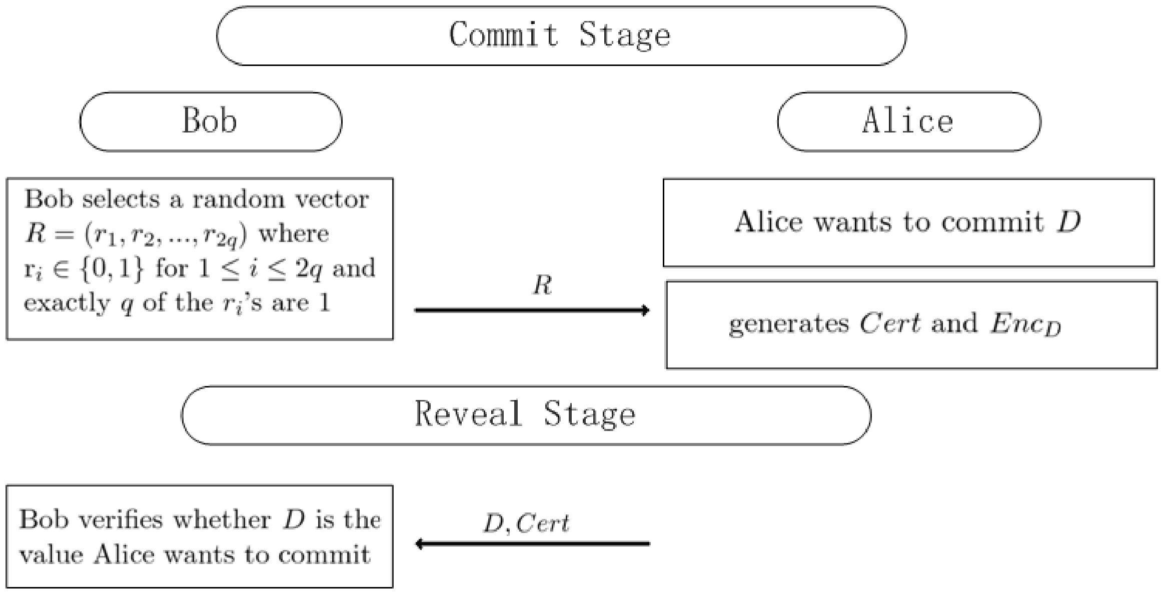

Commit Stage
Bob
Alice
Bob selects a random vector $R = (r_1, r_2, ..., r_{2q})$ where $r_i \in \{0,1\}$ for $1 \le i \le 2q$ and exactly q of the r_i's are 1
R
Alice wants to commit D
generates $Cert$ and Enc_D
Reveal Stage
Bob verifies whether D is the value Alice wants to commit
$D, Cert$

Figure 2.5: The protocol of Naor's construction (Naor, 1991)

Chapter 3

Secure and Trusted Verification for Honest Developers

The secure verification we are discussing in this chapter is two-fold: In Section 3.1 we will discuss the general case of secure and trusted verification and in Section 3.2, we will discuss our construction.

3.1 Secure and Trusted Verification for honest developers

In this section, we define a general secure and trusted verification scheme in which we do not specify what kind of information of a program is kept secret. Instead, we classify the information that should be kept secret from anyone except the developer as non-disclosed (secret) information and any other information as public information which can be known by everyone. This is because there are many different kinds of information that may need to be kept non-disclosed, such as certain parts of the whole implementation, or the relations between the modules of the program, or in one case the developer may want to hide the whole program while in some other cases it may want only to hide a module of the whole program. We want to make our definition of secure verification given in this section as general as it can be and include all cases above.

Moreover, apart from providing a method to hide the non-disclosed information, the verification scheme should also allow a developer to check whether the verifier evaluates the table graph correctly. A malicious or incompetent verifier may not only want to figure out the non-disclosed information but also warp the outcome of the verification. For instance, an incompetent verifier may claim that the table graph of a program does not pass the verification by a verification method, but what actually happens is that the verifier makes some mistakes during the evaluation of the table graph which causes the table graph's output to deviate from its expected output. Therefore, it is vital to require a method for a robust verification scheme to check the correctness of the verification done by the verifier.

Definition 16 and Figure 3.1 below describe what a secure verification scheme (VS) should be in the general case. VS includes an encryption algorithm $VS.\mathit{Encrypt}$ that outputs a secure version G' of a table graph G. The verifier verifies G' instead of G, and during the verification process, it may need to interact with the developer to correctly verify G'. In VS, if the verifier needs to interact with the developer, it sends a piece of information x to the developer who calls $VS.\mathit{Encode}(x)$ and returns $VS.\mathit{Encode}(x)$ to the verifier. After the evaluation finishes, whoever wants to check whether the verifier has correctly evaluated the table graph G' can use $VS.\mathit{Eval}$

together with a piece of public information (specified by $VS.Eval$) to find out.

Definition 16 (Secure and trusted verification)**.** *A secure and trusted verification scheme VS is a tuple of p.p.t. algorithms $(VS.Encrypt, VS.Encode, VS.Eval)$ such that*

(1) $VS.Encrypt(1^K, G)$ is a p.p.t. algorithm that takes the security parameter 1^K and the table graph G as input and outputs G'.

(2) $VS.Encode$ is a p.p.t. algorithm that takes an input x and returns an encoding Enc_x.

(3) $VS.Eval$ is a p.p.t. algorithm that takes as input $(1^k, Certificate)$ and outputs 1 or 0, where $Certificate$ is a piece of public information. $VS.Eval$ has a verifier V' hardcoded in it. V' is an honest verifier that satisfies Definition 7.

and VS satisfies Definition 17 and 18 below.

Definition 17 (Correctness)**.** *If $VS.Eval(1^K, Certificate) = 1$, then*

$$Pr_r[V(1^K, G, F_{spec}, VGA_r, CP)(r) = V(1^K, G', F_{spec}, VGA_r, CP)(r)] \geq 1 - negl(K), \quad (3.1)$$

$$Pr_r[V(1^K, G', F_{spec}, VGA_r, CP)(r) = V'(1^K, G', F_{spec}, VGA_r, CP)(r)] \geq 1 - negl(K), \quad (3.2)$$

where the probabilities are over the random bits r of V.

Definition 18 (Security)**.** *For two pairs of p.p.t. algorithms $A = (A_1, A_2)$ and $S = (S_1, S_2)$,*

1. *SI consist of any piece of information that the scheme VS wants to hide and all the information that can be known from it.*

2. *The oracle O in Exp^{ideal} below outputs $O(x) \notin SI$, for any input x that S_2 chooses.*

Consider the following two experiments:

```
Exp^{real}(1^K):
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', CP') \leftarrow VS.Encrypt(1^K, G, CP)$
3. $a \leftarrow A_2^{VS.Encode}(G', G, CP', CP, state_A)$
4. Output $a$
```

```
Exp^{ideal}(1^K):
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G'', CP'') \leftarrow S_1(1^K)$
3. $a \leftarrow A_2^{S_2^O}(G'', G, CP, CP'', state_A)$
4. Output $a$
```

VS is secure *if there exist a pair of p.p.t. simulators $S = (S_1, S_2)$ and an oracle O such that for all pairs of p.p.t. adversaries $A = (A_1, A_2)$, the following is true:*

$\forall$ *p.p.t. algorithm D,*

$$|Pr[D(Exp^{ideal}(1^K)_{K\in\mathbb{N}}, 1^K) = 1] - Pr[D(Exp^{real}(1^K)_{K\in\mathbb{N}}, 1^K) = 1]| \leq negl(K)$$

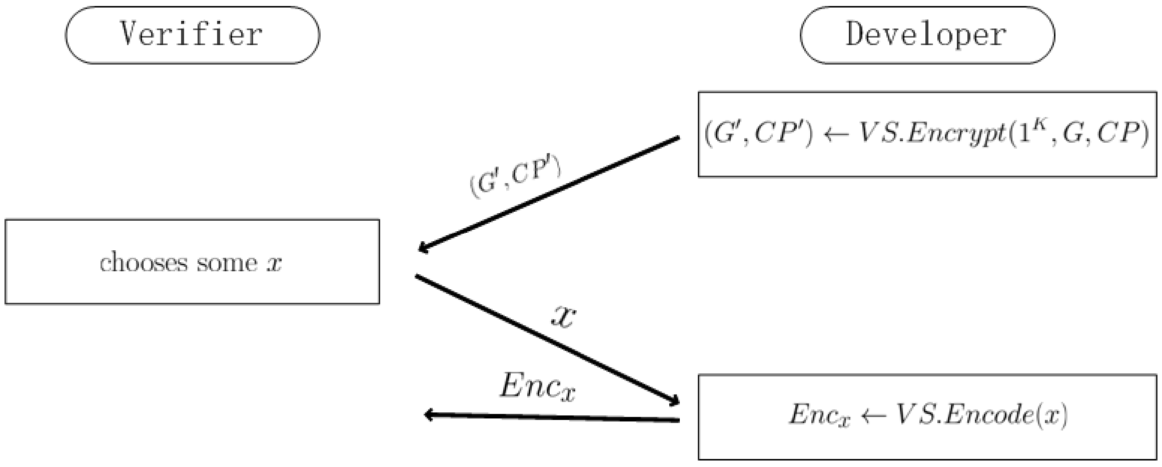

Verifier
Developer
$(G', CP') \leftarrow VS.Encrypt(1^K, G, CP)$
(G', CP')
chooses some x
x
Enc_x
$Enc_x \leftarrow VS.Encode(x)$

Figure 3.1: The protocol of VS in Definition 16

Remark 2. *In Step 3 of Exp^{real} in Definition 18, $VS.Encode$ is not an oracle of A_2. A_2 plays the role of a malicious verifier and A_2 interacts with the developer just like in Figure 3.1. A_2 asks the developer to run $VS.Encode$ with an input it chooses. Similarly, in Step 3 of Exp^{ideal} in Definition 18, S_2^O is also not an oracle of A_2. A_2 plays the role of a malicious verifier and A_2 interacts with the developer, who unlike in Exp^{real}, runs S_2^O instead of $VS.Encode$. Whenever we say A_2 queries $VS.Encode$ or A_2 queries S_2^O, we mean A_2 asks the developer to run $VS.Encode$ or S_2^O.*

Remark 3. *In Figure 3.1, the questions asked by the verifier and the answers replied by the developer are considered public knowledge known to everyone.*

Remark 4. *In Definition 16, the input of S_1 is the security parameter 1^K, plus any public information that S_1 needs to know.*

Remark 5. *The idea to use experiments such as Exp^{real} and Exp^{ideal} is from (Goldwasser* et al.*, 2013).*

Definition 19. *An* honest *developer is a developer that calls $VS.Encode(x)$ to generate Enc_x in Figure 3.1 for the x the verifier chooses.*

The definition above uses an assumption which we give below:

Assumption 4. *Given a table graph G, a developer always uses $VS.Encrypt$ to encrypt it as described in Definition 16.*

The above assumption implies that no matter whether the developer is honest or not, it always uses $VS.Encrypt$ to encrypt a table graph. For a dishonest developer who tries to hide bugs of its table graph, it can use a method M other than $VS.Encrypt$ to encrypt it. But doing this does not reduce the probability of detecting bugs from the encrypted table graph, because once the encrypted table graph is given to the verifier, it cannot be changed by the developer afterwards. Hence, suppose for a table graph G with a bug B, the verifier will find B after verifying an encrypted table graph G' generated by $VS.Encrypt$. Moreover the verifier will find no bug after verifying an encrytped table graph G'' generated by M. Then this means that the developer knows the existence of B and fixes it in G'' by using M. In our setting we need to point out that the verification is done on an encrypted table graph. Hence if the encrytped table graph G'' is really bug-free while the origianl table graph G has a bug B, then it is not the verification scheme's fault that the bug B is not detected. In any case, the developer can always release a buggy program after a bug-free encrypted table graph passes the verification.

Consequently, what we need to require from an honest developer is that it does not do anything to change the outcome of the verification process. If the verification is done by the verifier alone, then the developer cannot influence the verification outcome. However, due to the encryption of the table graph, interaction between the developer and the verifier is necessary. Essentially, the developer helps the verifier to finish the verification process via communicating with the verifier by running $VS.Encode$ to answer the verifier's questions. Therefore, this is the place where the developer can influence the verification process. If we require the developer to use $VS.Encode$, and not any other algorithms to answer the verifier's questions, then the developer has to honestly answer the questions of the verifier. Here, we assume that by using $VS.Encode$ to answer the verifier's questions, the verifier can finish the verification process with the outcome not influenced by the developer. This assumption is quite legitimate, because if $VS.Encode$ itself allows the developer to influence the outcome of the verification process, then anyone can figure this out from analysing $VS.Encode$ which is public to everyone.

3.2 Our construction

3.2.1 Problem description

In this section, we introduce a specific kind of secure verification scheme satisfying Definition 16. In our construction, we focus on protecting the content of a program. The content of a program is a broad notion, which in our setting includes the modules of a program and also the values of the intermediate results generated by evaluating the modules during the verification. In terms of an initial table graph, we aim at hiding the rows of the tables and the intermediate output values of the rows. When we are evaluating an initial table graph, only one row of a table can output values at the same time (due to disjointness property of tabular expressions) while the other rows will not output anything, and this piece of information is easily observed by the evaluator. Therefore, when evaluating a table graph G in Section 2.2.2, we only focus on hiding the content of the tables in G and the values of the intermediate outputs of these tables which are not $(\perp, \perp)$. The intermediate outputs that are $(\perp, \perp)$ can be exposed to the evaluator because these values only indicate that the corresponding rows in the initial table graph output nothing.

The reason to include the intermediate outputs as non-disclosed information is the following: The intermediate output of a table may reveal non-trivial information about the table itself. For example, a verifier may figure out that an intermediate table contains a constant function and even the constant itself, by observing the intermediate outputs of this table.

On the other hand, we consider the relations between different modules and the external input and output of the whole implementation as public knowledge which the verifier can have access to. In terms of the table graph G of the implementation, it is the structure graph G_{struc} of G and the external input/output X/Y of G. If we hide the graph structure as well as the table contents and the intermediate outputs, then what we can provide to the verifier is at most black box verification. On the contrary, in many occasions, knowing the graph structure alone facilitates verification process to a great extent. The verifier is able to select a path of G that it may want to check. Then it sends the path to the developer which returns an external input to G. The external input is chosen by the developer, but we do not specify how this input is chosen. The verifier when evaluating G with this external input, will evaluate the path it picks. Without exposing the graph structure G_{struc} to the verifier, this cannot be achieved.

3.2.2 Technique outline

According to Definition 16, the verification scheme consists of three algorithms VS.Encrypt, VS.Encode and VS.Eval. Below we are going to give our construction of the verification scheme which includes the above three algorithms and another algorithm VS.Path.

VS.Encrypt

First, we focus on developing a way to hide the content of every table in the table graph G while still allowing the verifier to somewhat evaluate the tables. We have already mentioned in Section 1.3 that we use fully homomorphic encryption to achieve this. We already know by the definition of FHE introduced in Section 2.5 that FHE enables evaluation of any computable function which takes fully homomorphically encrypted data as input. The encrypted data will not leak to the evaluator during evaluation. Hence, by using FHE, we achieve computing sensitive data while still protecting them from leaking to anyone who computes them. However, our goal is to hide tables, which are functions and not mere data. Thus simply applying FHE is not an option. Instead, we borrow the idea of computing encrypted functions with universal circuits (Sander *et al.*, 1999). We briefly illustrate this idea bellow.

If we can build a universal circuit U that takes a circuit C and the circuit's input x as its input, and outputs the circuits' output $C(x)$, then we can apply certain encryption scheme (which supports evaluation over encrypted data) to encrypt C and

x and get the ciphertext C', x'. Then we evaluate U with C', x' as its input and output an encryption of $C(x)$. Finally we decrypt it and get $C(x)$. The prerequisite of this idea is that the universal circuit actually exists. From (Sander *et al.*, 1999) (Valiant, 1976) we know that there is a universal circuit U for any circuit C of depth d and size s such that U takes as input a string S_C (which can be efficiently computed from C) and a value x (which is an input of C) and outputs C(x). Namely, $U(S_C, x) = C(x)$ (U is parameterized by d, s).

The idea above by Sanders et al. successfully evaluates $C(x)$ with its input x while nothing about C is leaked to the evaluator other than $C(x)$. Meanwhile, FHE perfectly fits into the idea which supports evaluation of arbitrary function over encrypted data. (This idea of using FHE to encrypt a program and treat it as an input to a universal input is also briefly mentioned in (Goldwasser *et al.*, 2013).)

Therefore, what we do in our construction is to convert the rhs function F_i of a table PT_i to a circuit C_i and build a universal circuit U as well as the string S_{C_i} of C_i. For an input x to PT_i, we fully homomorphically encrypt S_{C_i} and x with FHE's public key hpk and get E_{C_i}, x'. Consequently we can construct an encrypted version of PT_i, which is $PT_i' = (True, E_{C_i})$. If a verifier wants to evaluate PT_i' with fully homomorphically encrypted input x', it just runs $FHE.Eval(hpk, U, E_{C_i}, x')$ and gets the output $PT_i'(x') = FHE.Eval(hpk, U, E_{C_i}, x')$. Because $FHE.Eval(hpk, U, E_{C_i}, x') = FHE.Enc(hpk, C(x))$, we know $PT_i'(x') = FHE.Enc(hpk, C(x))$. In conclusion, by using the above idea, for a table PT_i with an input x chosen by the verifier, the verifier can correspondingly evaluate the FHE encrypted version PT_i' with x' (which is the FHE encryption of input x) and get an output which is the FHE encryption of $PT_i(x)$.

Second, suppose for every table PT_i in table graph G, we have the corresponding secure version table PT'_i. Then we have a new table set $TS' = \{PT'_1, ..., PT'_n\}$ from $TS = \{PT_1, ..., PT_n\}$, which is the table set of G. As mentioned before, because our construction focuses on hiding the content of the tables in TS, we do not mind giving out the structure graph G_{struc} of G to the verifier. Thus for the new secure version table graph G' that the verifier will finally verify, $G'_{struc} = G_{struc}$.

VS.Encode

Having $VS.Encrypt$, is not enough for our verifications scheme to work. The reason is as follows. Suppose the verifier is evaluating a table PT'_i whose output is an external output. From the construction of PT'_i we know that the output of PT'_i is actually an FHE encrypted ciphertext. But the verifier needs to know the plaintext of this ciphertext in order for verification to work. Moreover, we certainly cannot allow the verifier itself to decrypt this ciphertext, because otherwise the verifier would be able to decrypt the encrypted circuit inside PT'_i as well, and the security of the scheme is compromised. Consequently, the verifier needs to ask the developer who calls an algorithm $VS.Encode$ to decrypt this ciphertext for the verifier.

On the other hand, simply asking the developer to decrypt the encrypted output does not work for intermediate outputs. Suppose the verifier is working on G and gets an intermediate output y_i of a table PT_i in G. If y_i is not $(\perp, \perp)$, then this y_i will be the input of another table PT_j in G. (Assume there is a corresponding edge pointing from PT_i to PT_j.) If y_i is $(\perp, \perp)$, then this y_i should not be the input to any other table in G. Knowing whether y_i is $(\perp, \perp)$ is crucial to a correct evaluation, and thus important to the verifier. Now consider the case where PT_i is converted into a secure version table PT_i' and the intermediate output y_i' of PT_i' hides all the information about y_i. Then the verifier certainly cannot figure out whether y_i is $(\perp, \perp)$ or not from y_i'. Hence we need $VS.Encode$ to be able to indicate whether the corresponding y_i of y_i' is $(\perp, \perp)$ or not.

Moreover, consider the case where the verifier wants to evaluate a table $PT_i' = (True, E_{C_i})$ which has an external input x. Suppose the verifier chooses the external input x to be $(\top, a)$. Then the verifier is required to provide an FHE encryption x' of x to evaluate PT_i', because the evaluation of PT_i' is essentially running $FHE.Eval$ with hpk, E_{C_i}, U as well as an FHE encrypted input x'. Therefore, a simple way here is to let the verifier do the FHE encryption of $(\top, a)$ and get $FHE.Enc(hpk, (\top, a))$. Then the verifier can evaluate PT_i' as follows:

$$PT_i'(FHE.Enc(hpk, (\top, a))) \leftarrow FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, (\top, a))).$$

However, this simple solution has security problems which can be exploited by a malicious verifier. The verifier can choose an intermediate output and claim this intermediate output to be the encryption of an external input that it has chosen. Then, through the interaction with $VS.Encode$, the verifier may figure out partial information about this intermediate output. Therefore, in order to prevent the verifier from doing this, we employ $VS.Encode$: For any external input the verifier chooses, the verifier cannot fully homomorphically encrypt the input by itself; It must send the input to the developer who by calling $VS.Encode$, generates the FHE encryption of this value.

VS.Eval

VS.Eval is a publicly accessible algorithm that allows anyone to check if the verifier has done the evaluation of the enrypted table graph G' correctly. In order to achieve this, the interaction between the verifier and the developer has to be public. In our construction we have a public file QA_E that records the questions asked by the verifier and the corresponding answers generated by the developer by calling *VS.Encode*. By reading QA_E, *VS.Eval* can know what input the verifier evaluates with a particular table and what output the verifier gets. Then *VS.Eval* can evaluate the table again with the same input and check whether the output is the same.

VS.Path

As mentioned in subsection 3.2.1, our construction is expected to provide the ability to allow the verifier to evaluate paths that itself chooses. This ability is not necessarily included in Definition 16 due to the generality of the definition. In our construction, when the verifier wants to evaluate a path of a table graph, the developer runs *VS.Path* with the path chosen by the verifier and generates an external input such that when used, it will cause the evaluation of the tables on the path chosen by the verifier.

Conversion of functions to circuits

We have mentioned above that in our construction the rhs functions in the tables of a table graph will be converted into boolean circuits in order for our construction to work properly. But we did not say how the conversion is done or whether this conversion is practical. Essentially what our construction requires is an automatic way to transform higher level description of a function into a boolean circuit. Transformation of a higher level description of a function into a lower level description like a circuit was done in the work of (Malkhi *et al.*, 2004) and (Huang *et al.*, 2011). (Malkhi *et al.*, 2004) introduces a system called FAIRPLAY which includes a high level language SFDL that is similar to C and Pascal. FAIRPLAY allows the developer to describe the function in SFDL, and automatically compiles it into a description of a boolean circuit. A similar system is described in (Huang *et al.*, 2011), which is faster and more practical than FAIRPLAY. This system allows the developer to design a function directly in Java, and automatically compiles it into a description of a boolean circuit.

3.2.3 Construction

In this section we give our construction of a secure and trusted verification scheme.

VS.Encrypt (see Algorithm 1) takes as input the security parameter 1^K and the table graph G, and outputs a secure version table graph G' (also called an *encrypted* table graph), an FHE public key hpk and a universal circuit set U.

VS.Encode (see Algorithm 2) answers two kinds of questions:

(1) The first kind of question q_1 : $VS.Encode$ takes $(i, x^i_j, null, q_1)$ as input and outputs $VS.Encode(i, x^i_j, null, q_1)$. The index i indicates the ith table $PT'_i \in G'$ and x^i_j denotes an external input to PT'_i.

(2) The second kind of question q_2 : $VS.Encode$ takes $(i, x^i, PT'_i(x^i), q_2)$ as input and outputs $VS.Encode(i, x^i, PT'_i(x^i), q_2)$. The index i indicates the ith table $PT'_i \in G'$ and x^i denotes the input to PT'_i.

Algorithm 1 VS.Encrypt$(1^K, G)$

```
1: $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$
2: Construct a universal circuit $U$ such that for any circuit $C$ of size $s$ and depth $d$ and a corresponding string $S_C$ (which can be efficiently computed from knowing $C$), $U(S_C, x) = C(x)$.
3: Suppose $C$ outputs $m$ bits. Construct $m$ circuit $U_1, ..., U_m$ such that for any $i \in [m]$, $U(x, S_C)$ outputs the $i$th bit of $U(x, S_C)$.
4: for all $PT_i \in TS, i \in \{1, ..., n\}$ do
5:     construct $C_i$ such that $C_i(x^i) = PT_i(x^i)$
6:     construct a string $S_{C_i}$ from $C_i$
7:     $E_{C_i} \leftarrow FHE.Encrypt(hpk, S_{C_i})$
8:     $PT'_i \leftarrow (True, E_{C_i})$
9: end for
10: $TS' \leftarrow \{PT'_1, ..., PT'_n\}$
11: $G'_{struc} \leftarrow G_{struc}$
12: return $G' = [TS', G'_{struc}], hpk, U = (U_1, ..., U_m)$
```

$VS.Encode$ has a memory S.

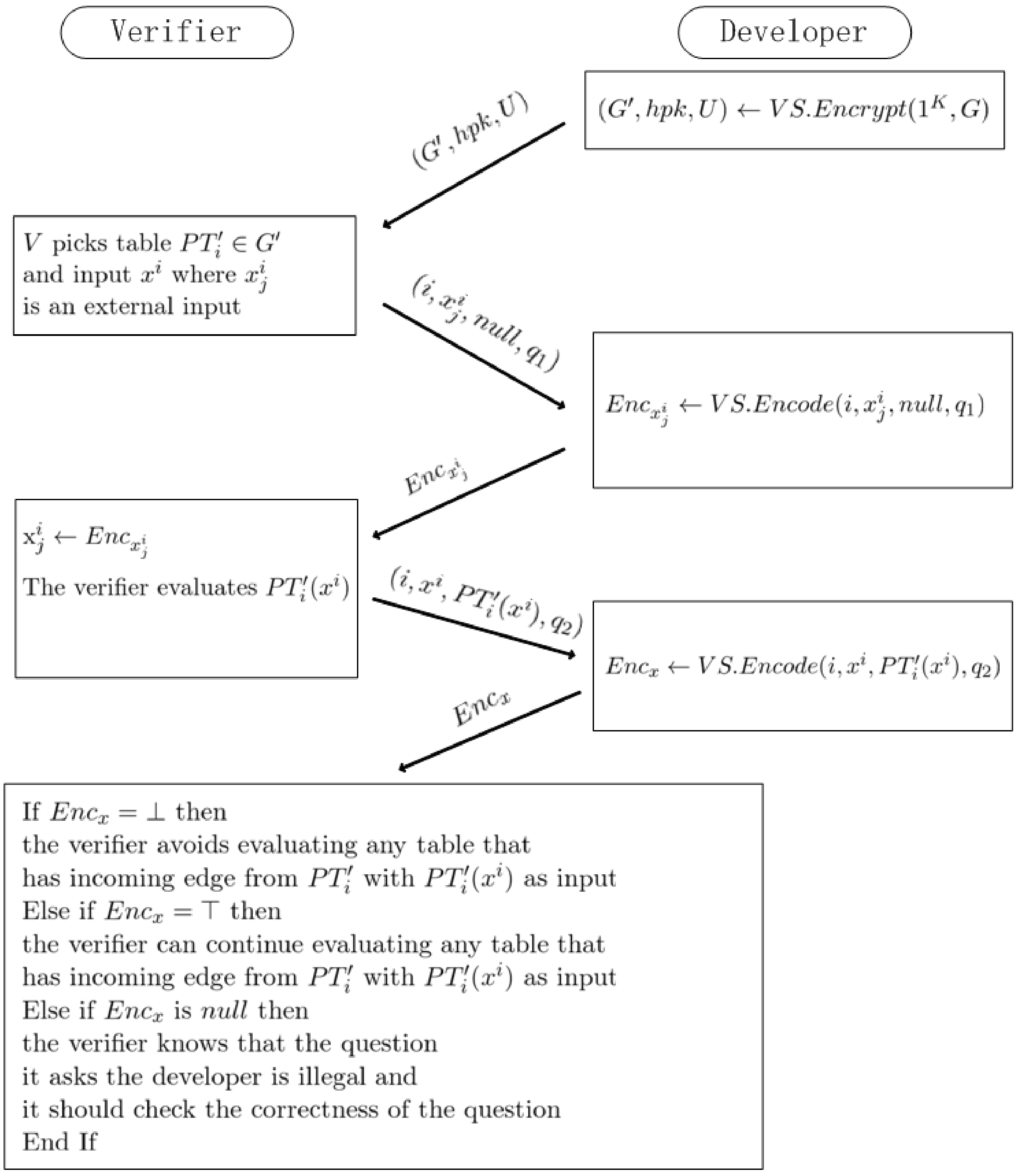
VS.Eval (see Algorithm 3) takes as input (1^K,QA_E,G',hpk,U) and outputs 1 or 0. $QA_E = \{(Q_1, A_1), ..., (Q_n, A_n)\}$. For any pair $(Q_i, A_i) \in QA_E$, Q_i is a question asked by the verifier and A_i is the answer generated by the developer running $VS.Encode(Q_i)$. QA_E is generated during V's verification of G' ($V^{VS.Encode}(1^K, G', F_{spec}, CP)$).

VS.Path (see Algorithm 4) takes as input $(PT'_{i_1}, ..., PT'_{i_p})$, and outputs an external input X to the table graph G.

If the verifier wants to evaluate a path $P = u_{i_1} \rightarrow u_{i_2} \rightarrow ... \rightarrow u_{i_p}$, it asks the developer to run $VS.Path(PT'_{i_1}, ..., PT'_{i_p})$ and $VS.Path$ outputs an external input X which will be returned by the developer to the verifier. X will allow the verifier to evaluate P. According to the construction of $VS.Path$, we know that evaluating G with X will also include evaluation of P.

Remark 6. *For $VS.Path$ we make the following assumption: If the verifier chooses a path P when verifying G' and $VS.Path(P) = X$, then when the verifier chooses the same path P to evaluate when verifying G, it will use the same external input X.*

The protocol of VS (see Figure 3.2) describes what the developer and the verifier should do in VS.


Verifier
Developer
$(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
(G', hpk, U)
V picks table $PT_i' \in G'$ and input x^i where x_j^i is an external input
$(i, x_j^i, null, q_1)$
$Enc_{x_j^i} \leftarrow VS.Encode(i, x_j^i, null, q_1)$
$Enc_{x_j^i}$
$x_j^i \leftarrow Enc_{x_j^i}$
The verifier evaluates $PT_i'(x^i)$
$(i, x^i, PT_i'(x^i), q_2)$
$Enc_x \leftarrow VS.Encode(i, x^i, PT_i'(x^i), q_2)$
Enc_x

```
If $Enc_x = \perp$ then
    the verifier avoids evaluating any table that
    has incoming edge from $PT_i'$ with $PT_i'(x^i)$ as input
Else if $Enc_x = \top$ then
    the verifier can continue evaluating any table that
    has incoming edge from $PT_i'$ with $PT_i'(x^i)$ as input
Else if $Enc_x$ is $null$ then
    the verifier knows that the question
    it asks the developer is illegal and
    it should check the correctness of the question
End If
```

Figure 3.2: The protocol of VS in our implementation

Remark 7. *In our implementation, the evaluation of a table $PT_i'(x^i)$ where $PT_i' \in G'$*

is done as follows:

$$PT'_i(x^i) \leftarrow FHE.Eval(hpk, U, E_{C_i}, x^i),$$

where hpk, U, E_{C_i} belong to the output of $VS.Encrypt$ in Algorithm 1.

Remark 8. *The* evaluation of an encrypted table graph G' *is a little bit different than evaluation of a normal table graph G which is described in section 2.3. The evaluation of G' with an external input to G' (denoted as $G'(X)$) is described below in Algorithm 5.*

3.2.4 An example of applying condition/decision coverage verification to our construction

In this subsection we use an example in Figure 3.3 to show how to apply our verification scheme to actually verify a specific table graph. In Figure 3.3 there is an initial table graph that is to be verified by the verifier V.

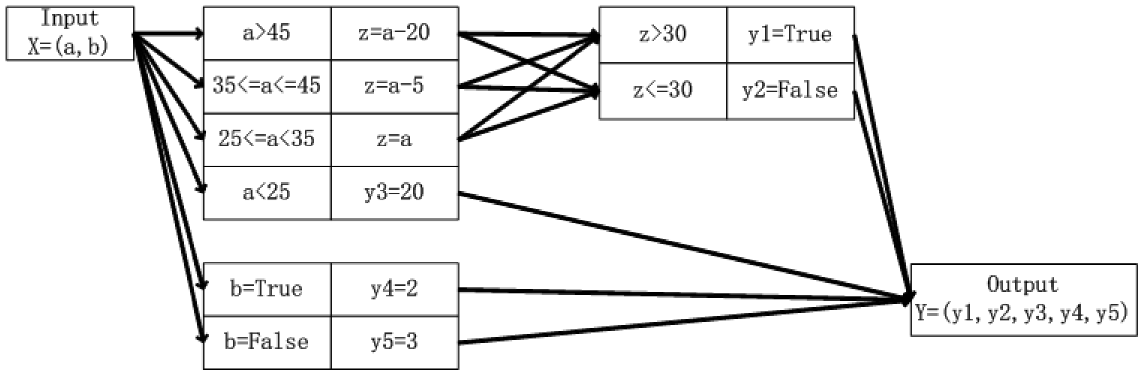


Figure 3.3: An initial table graph $\mathcal{G}$ of an implementation

First the developer transforms this initial table graph into a table graph G introduced in Section 2.2 (see Figure 2.3). Then

$$F_1(a) = \begin{cases} (\top, a-20) & \text{, if } a > 45 \\ (\bot, \bot) & , otherwise \end{cases} \qquad F_2(a) = \begin{cases} (\top, a-5) & \text{, if } 35 <= a <= 45 \\ (\bot, \bot) & , otherwise \end{cases}$$

$$F_3(a) = \begin{cases} (\top, a) & \text{, if } 25 <= a < 35 \\ (\bot, \bot) & , otherwise \end{cases} \qquad F_4(a) = \begin{cases} (\top, 20) & \text{, if } a < 25 \\ (\bot, \bot) & , otherwise \end{cases}$$

$$F_5(z) = \begin{cases} (\top, True) & \text{, if } z > 30 \\ (\bot, \bot) & , otherwise \end{cases} \qquad F_6(z) = \begin{cases} (\top, False) & \text{, if } z <= 30 \\ (\bot, \bot) & , otherwise \end{cases}$$

$$F_7(b) = \begin{cases} (\top, 2) & \text{, if } b = True \\ (\bot, \bot) & , otherwise \end{cases} \qquad F_8(b) = \begin{cases} (\top, 3) & \text{, if } b = False \\ (\bot, \bot) & , otherwise \end{cases}$$

The developer applies our content-secure verification scheme VS to G. It runs $VS.Encrypt$ as follows. $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$. Figure 3.4 is the encrypted table graph G'.

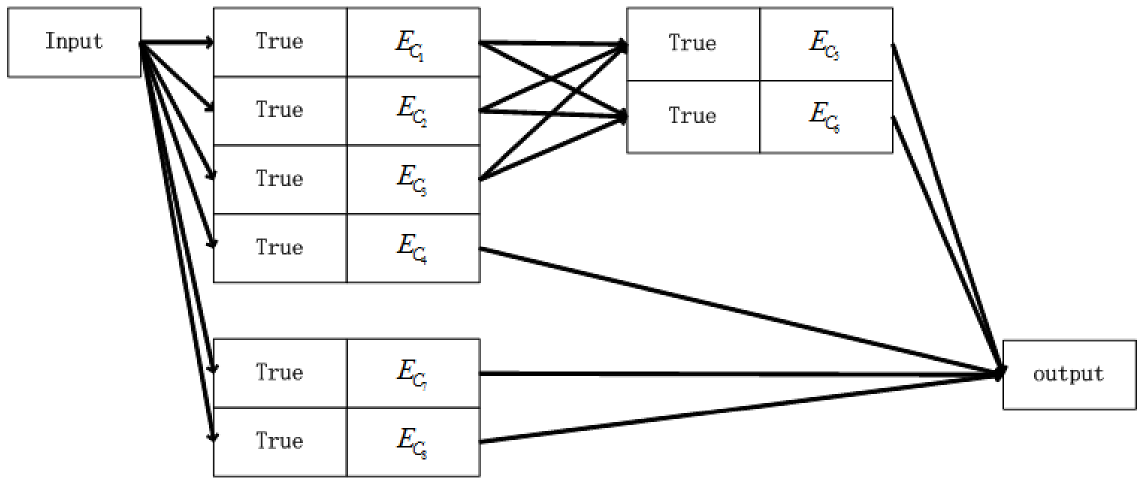


Figure 3.4: An encrypted table graph G' after applying VS to G

According to Figure 3.2, the verifier V receives G' and does the verification on G'. We show how V can do MC/DC verification (Hayhurst *et al.*, 2001) on G'. MC/DC performs structural coverage analysis. First it gets test cases generated from analysing a given program's requirements. Then it checks whether these test cases actually covers the given program's structure and finds out the part of the program's structure which is not covered. First we assume the VGA that V uses will do MC/DC verification after it generates the test cases. Suppose V runs VGA to generate the test cases, based on requirements-based tests (by analysing F_{spec}), and these test cases are stored in EI. Then V picks an external input X to G' from EI and starts evaluating G' with X.

For $X = (a = 26, b = True)$, and according to Figure 3.2, V sends the following queries to the developer DL (The queries are in the format of the input of $VS.Encode$): $Q_1 = (1, 46, null, q_1)$, $Q_2 = (2, 46, null, q_1)$, $Q_3 = (3, 46, null, q_1)$, $Q_4 = (4, 46, null, q_1)$, $Q_5 = (7, True, null, q_1)$, $Q_6 = (8, True, null, q_1)$, because it needs to evaluate PT'_1, PT'_2, PT'_3, PT'_4, PT'_7, PT'_8 as well as an encoding for the external inputs of each table. We take the evaluation of the path (Input $\rightarrow PT'_1 \rightarrow PT'_5 \rightarrow$ Output) as an example. For query Q_1, DL evaluates $VS.Encode(Q_1)$ and returns $FHE.Enc(hpk, 46)(FHE.Enc(hpk, 46)$ is the output of $VS.Encode(Q_1))$, which is the input x^1 to PT'_1. Then V runs $FHE.Eval(hpk, U, x^1, E_{C_1})$ and outputs $PT'_1(x^1)$. After this V sends $(1, x^1, PT'_1(x^1), q_2)$ to DL. Because we know that for $PT_1 \in G$, if 46 is the input to PT_1, then the output will be $(\top, 26)$. Thus for the query $(1, x^1, PT'_1(x^1), q_2)$, DL evaluates $VS.Encode(1, x^1, PT'_1(x^1), q_2)$ and returns $\top$. Hence V knows that for a=46 as an external input, the lhs predicate (a decision and condition) of $PT_1 \in G$ is satisfied, and the rhs function of $PT_1 \in G$ is covered.

After finishing evaluating PT'_1, V starts evaluating PT'_5 and PT'_6 with $PT'_1(x^1)$ as their input. $x^5 = PT'_1(x^1)$ is PT'_5's input. After finishing evaluating PT'_5, V gets $PT'_5(x^5)$ as the output. Then V sends $(5, x^5, PT'_5(x^5, q_2)$ to DL. DL evaluates $VS.Encode(5, x^5, PT'_5(x^5, q_2)$ and $VS.Encode$'s output is $True$ (Because $(46 > 30)$, PT_5's output is $(\top, True)$. We also know that the output of PT'_5 is an external output. Accordingly, $VS.Encode$ outputs $True$). Therefore, DL returns $True$ to V. Then V knows that $y1 = True$ as well as the fact that the lhs predicate (a decision and condition) of $PT_5 \in G$ is satisfied and the rhs function of $PT_5 \in G$ is covered.

After evaluating G' with $X = (a = 26, b = True)$ by similar steps as described above and getting the external output $Y = (True, \bot, \bot, 2, \bot)$, V knows that for X, the lhs predicates of PT_1, PT_5 and PT_7 are satisfied while the rest tables' lhs predicates are not satisfied. Hence, V knows that the predicates of PT_1, PT_5 and PT_7 are $True$ while the predicates in the rest tables of G are $False$ and the statements (the rhs functions of the tables in G) of PT_1, PT_5 and PT_7 are covered. Moreover,

V compares Y with $F_{spec}(X)$ to see if G behaves as expected with X as an external input.

V will keep evaluating G' with the rest external inputs in EI, and by interacting with DL in the way as described above, it does the structural coverage analysis of the requirements-based test cases. He will be able to know whether the external inputs in EI covers every predicates in G. Additionally, it will be able to know whether G behaves as expected in the requirements specification described by F_{spec}.

3.2.5 Proof of the implementation correctness and security

For our implementation of VS in subsection 3.2.3, $VS.Encrypt$, $VS.Encode$, S_1 and S_2 are the implementation of $VS.Encrypt$, $VS.Encode$, S_1 and S_2 of Definition 18. In our implementation $CP' = CP'' = CP$. The input $Cerificate$ of $VS.Eval$ in Definition 20 in our implementation is QA_E, G', hpk, U.

In Definition 18 we have two simulators S_1 and S_2 simulating $VS.Encrypt$ and $VS.Encode$. S_1 takes any public information it needs. In our implementation we add another algorithm $VS.Path$, so we add another simulator S_3 to simulate $VS.Path$. The input to simulator S_1 consists of the circuit size s and depth d of all the circuits resulting from rhs functions in the tables in G, as well as the structure graph G_{struc} of G (The construction of the simulators S_1, S_2 and S_3 can be found in the proof of Theorem 2). S_2's simulator oracle O_1 in our implementation is the oracle O in Definition 18. We add one oracle O_2 for S_3, which is not in Definition 18.

In our proof below, FHE is the fully homomorphic encryption scheme introduced in Section 2.5.

Proof of the implementation correctness

First we prove that our implementation satisfies Definition 17.

Theorem 1. *The verification scheme VS introduced in section 3.2 satisfies Definition 17.*

Proof. We will need the following lemmas:

Lemma 1. *When $VS.Eval$ outputs 1, for any table $PT'_r \in TS'$ and x^r, the output y claimed by V as $PT'_r(x^r)$ must be equal to $PT'_r(x^r)$. That is, V* evaluates every table correctly.

Proof. $VS.Eval(1^K, QA_E, G', hpk, U)$ checks whether y is equal to $PT'_r(x^r)$ in line 3 of Algorithm 3. (For any $(Q_i, A_i) \in QA_E$ where $Q_i = (r, x^r, y, q_2)$, $VS.Eval$ evaluates $PT'_r(x^r)$ and checks whether y is equal to $PT'_r(x^r)$.) □

From now on we concentrate on the subset of QA_E that is consistent (see Definition 5). For non-consistent QA_E's, Algorithm 2 (lines 13,14,16) will return *null* evaluations, and, therefore, security and correctness are not an issue.

Lemma 2. *Given the external input set X to both G' and G, for any table $PT_i \in TS$ with input x^i and output $PT_i(x^i)$, table $PT'_i \in TS'$ and its input w^i satisfy the following:*

$$w^i = FHE.Enc(hpk, x^i), PT'_i(w^i) = FHE.Enc(hpk, PT_i(x^i)), \tag{3.3}$$

where hpk is the public key of FHE that is generated as follows:

$$(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$$

Proof. We prove this lemma by induction on the level k of a table.

(1) Base case: $k = 1$. For any level 1 table $PT_i \in TS$ where x^i is its input, $PT'_i \in TS'$ and its input w^i satisfy the following:

$$w^i = FHE.Enc(hpk, x^i), PT'_i(w^i) = FHE.Enc(hpk, PT_i(x^i)).$$

According to the construction of $VS.Encrypt$, $G'_{struc} = G_{struc}$ and $PT'_i(w^i) = FHE.Eval(hpk, U, E_{C_i}, w^i)$, where E_{C_i} is the rhs function of PT'_i (see Algorithm 1). Also, according to the construction of $VS.Encode$, $w^i = FHE.Enc(hpk, x^i)$. Hence

$$\begin{aligned} PT'_i(w^i) =& FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i)) \\ =& FHE.Enc(hpk, PT_i(x^i)) \end{aligned}$$

(The second equation is valid because, according to (2.1),

$$\begin{aligned} PT_i(x^i) &= C_i(x^i) = U(S_{C_i}, x^i) \\ &= FHE.Dec(hsk, FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i))), \end{aligned}$$

where C_i, U, S_{C_i} are in Algorithm 1.

Hence,

$$\begin{aligned} FHE.Enc(hpk, PT_i(x^i)) = FHE.Enc(&hpk, \\ &FHE.Dec(hsk, FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i)))) \\ &= FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i)). \end{aligned}$$

)

(2) Inductive step: Suppose that for any table $PT_i \in TS$ whose level is smaller than $k+1$ and x^i is its input, the level k table $PT'_i \in TS'$ and its input w^i satisfy the following:

$$w^i = FHE.Enc(hpk, x^i), PT'_i(w^i) = FHE.Enc(hpk, PT_i(x^i)).$$

Then we show that for any level $k+1$ table $PT_j \in TS$ where x^j is its input, the level $k+1$ table $PT'_j \in TS'$ and its *encoded* input w^j satisfy the following:

$$w^j = FHE.Enc(hpk, x^j), PT'_j(w^j) = FHE.Enc(hpk, PT_j(x^j))$$

For PT_j, its table input $x^j = [x^j_1, x^j_2, ..., x^j_s]$ satisfies the following: For any $x^j_p \in x^j$, either $x^j_p = PT_{i_p}(x^{i_p})$, or $x^j_p \in X$. In case $x^j_p \in X$, according to Definition 3, we can treat it as a product of a virtual level 0 table.

Since PT_j is a level $k+1$ table, for any $x^j_p \in x^j$, $x^j_p = PT_{i_p}(x^{i_p})$, where PT_{i_p} is a table whose level is smaller than $k+1$. Then according to the inductive hypothesis, for w^j which is the table input of PT'_j,

$$w^j_p = PT'_{i_p}(w^{i_p}) = FHE.Enc(hpk, PT_{i_p}(x^{i_p})) = FHE.Enc(hpk, x^j_p).$$

For any $x^j_p \in X$, the corresponding $w^j_p = FHE.Enc(hpk, x^j_p)$. Hence,

$$\begin{aligned} w^j =& [w^j_1, ..., w^j_s] \\ =& [FHE.Enc(hpk, x^j_1), ..., FHE.Enc(hpk, x^j_s)] \\ =& FHE.Enc(hpk, x^j) \end{aligned}$$

(The last equation is valid because $FHE.Enc$ encrypts a string bit by bit.

$$\begin{aligned}FHE.Enc(hpk, x^j) =& FHE.Enc(hpk, [x_1^j, x_2^j, ..., x_s^j])\\ =& [FHE.Enc(hpk, x_1^j), ..., FHE.Enc(hpk, x_s^j)]\end{aligned}$$

)

Since we already know that $w^j = FHE.Enc(hpk, x^j)$, we have

$$\begin{aligned}PT'_j(w^j) =& FHE.Eval(hpk, U, E_{C_j}, FHE.Enc(hpk, x^j))\\ =& FHE.Enc(hpk, PT_j(x^j))\end{aligned}$$

similarly to the base case.

□

Lemma 3. *The input-output functionality of G' is the same as the input-output functionality of G.*

Proof. Given the external input set X to both G' and G, suppose $PT_{i_1}, ..., PT_{i_s} \in TS$ are the output level tables that are actually evaluated. Suppose their corresponding outputs are $y_1, ..., y_s$. Then from Lemma 2 we know that $PT'_{i_1}, ..., PT'_{i_s} \in TS'$ satisfy the following:

$$PT'_{i_1}(w^{i_1}) = FHE.Enc(hpk, y_1), ..., PT'_{i_s}(w^{i_s}) = FHE.Enc(hpk, y_s).$$

Accordingly, for every $j \in [s]$, by asking the developer to call

$$VS.Encode(i_j, w^{i_j}, PT'_{i_j}(w^{i_j}), q_2),$$

V gets $y_1, ..., y_s$ as a reply (see line 29 in Algorithm 2). Hence the input-output functionality of (T'_{des}, G'_{des}) is the same as the input-output functionality of (T_{des}, G_{des}). $\square$

The following lemma applies to paths of G', but can be easily extended to any consistent subgraph of G' (e.g. in topological order).

Lemma 4. *For an encrypted table graph G', V asks the developer to call $VS.Path(PT'_{i_1}, ..., PT'_{i_p})$. $VS.Path$ outputs an external input X such that evaluating G' with X as the external input will include a successful evaluation of path $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$.*

Proof. According to the construction of $VS.Path$ (see Algorithm 4), we know that the evaluation of G with X as input includes the evaluation of tables $PT_{i_1}, ..., PT_{i_p}$. The inputs to $PT_{i_1}, ..., PT_{i_p}$ are $x^{i_1}, ..., x^{i_p}$. Since V evaluates these tables, $x^{i_1}, ..., x^{i_p}$ cannot contain a *null* or $(\perp, \perp)$. Therefore, w^{i_k} cannot contain *null* or the FHE encryption of $(\perp, \perp)$. According to Lemma 2, if V evaluates G' with X, then for the corresponding tables $PT'_{i_1}, ..., PT'_{i_p}$, the inputs $w^{i_1}, ..., w^{i_p}$ to these tables will satisfy (3.3), and $PT'_{i_1}, ..., PT'_{i_p}$ can be evaluated, because of (3.3). Since during the evaluation of G' with X, V evaluates all the tables that can be evaluated, path $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$ will also be evaluated. $\square$

Lemma 2 and hence Lemma 3 are based on the assumption that the verifier evaluates every table correctly and also follows the consistency requirement in Definition 5. When $VS.Eval$ outputs 1, V satisfies Lemma 1. Hence, when V evaluates G', the output of G' is the same as the output of evaluating G with the same inputs.

From the assumption in Remark 6, we know that all external inputs generated during $V(1^K, G', F_{spec}, CP, VGA)$ and $V(1^K, G, F_{spec}, CP, VGA)$ must be the same. Hence, we know that (3.1) in Definition 17 is true.

Moreover, $VS.Eval$ knows the VGA V uses as well as the random bits of the VGA, so from reading QA_E, $VS.Eval$ can check whether V satisfies (3.2) in Definition 17 (see Definitions 16 and 17). If $VS.Eval(1^K, QA_E, G', hpk, U)$ outputs 1, (3.2) is satisified (see lines 9 - 18 in Algorithm 3).

Thus, Theorem 1 is proved. □

Proof of the implementation security

Theorem 2. *The verification scheme VS introduced in section 3.2 satisfies Definition 18.*

Proof. We construct a tuple of simulators (S_1, S_2, S_3) which satisfies Definition 16 as follows:

(1) Let s be the common circuit size and d the common depth of all the circuits converted from rhs functions in the tables in G. $S_1(1^K, s, d, G_{struc})$ will generate a table graph $\tilde{G} = (\tilde{TS}, G_{struc})$ such that $\forall \tilde{PT}_i \in \tilde{TS}$, $\tilde{PT}_i = (True, \tilde{F}_i)$, where $\tilde{F}_i$ is a function that can be converted into a size s, depth d circuit that outputs m bits. S_1 runs $VS.Encrypt(1^K, \tilde{G})$ (Algorithm 1) and obtains G'', hpk, U.

(2) S_2 receives queries from A_2 in Definition 18 and queries oracle O_1 which is described in Algorithm 6. ($S_2^{O_1}$ simulates $VS.Encode$. The queries from A_2 are inputs to $VS.Encode$.) O_1 has a state $[S, H, id]$ which contains two initially empty sets S and H and a global variable id initially set to 0. S_2 returns the output of O_1 as answers to the queries of A_2.

(3) S_3 receives queries from A_2, and queries oracle O_2 which is described in Algorithm 7. S_2 then returns the output of O_2 to A_2. ($S_3^{O_2}$ simulates $VS.Path$. The queries from A_2 are inputs to $VS.Path$.)

First, we construct an experiment Exp which will be used in the following. Exp is slightly different from Exp^{real} in Definition 18. The only difference is that in Exp, the queries of A_2 are not answered by calls to $VS.Encode$ and $VS.Path$ as in Exp^{real}. Instead, they are answered by oracles O'_1 and O'_2 which work the same way as $S_2^{O_1}$ and $S_3^{O_2}$. ($S_2^{O_1}$ is in the simulated experiment Exp^{ideal} of Definition 18 and $S_3^{O_2}$ is added to Exp^{ideal} to be an oracle for A_2.) The details are shown below:

```
$Exp(1^K)$ with any p.p.t. $A_1, A_2$:
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
3. $a \leftarrow A_2^{O'_1, O'_2}(1^K, G', G, CP, hpk, U, state_A)$
4. Output a
```

Lemma 5. *Exp^{real} and Exp are computationally indistinguishable.*

Proof. We know that if $VS.Encode$ and O'_1 have the same input-output functionality, and $VS.Path$ and O'_2 have the same input-output functionality, then Lemma 5 must be true. First we prove that $VS.Encode$ and O'_1 have the same input-output functionality.

O'_1 works the same as $S_2^{O_1}$. From the construction of $VS.Encode$ (see lines 1 - 9 in Algorithm 2) and O'_1 (see lines 1 - 11 in Algorithm 6), we know that for a query

(i, u, v, q_1), both algorithms will output the FHE encryption of u. Therefore they have the same input-output functionality.

For a query (i, u, v, q_2), there are two cases.

Case 1: $VS.Encode(i, u, v, q_2)$ outputs *null*. It is easy to see that $O'_1(i, u, v, q_2)$ will also output *null*. To see this, note that the following situations in Algorithm 2 will cause $VS.Encode(i, u, v, q_2)$ to output *null*:

(1) For an intermediate input $x^i_j \in u$ which should be the output of PT'_k, x^i_j is not PT'_k's output (see line 13).

(2) For an intermediate input $x^i_j \in u$ which should be the output of PT'_k, x^i_j is PT'_k's output, but PT'_k's output is $(\perp, \perp)$ (see line 14).

(3) For an external input $x^i_j \in u$, x^i_j is not an FHE encryption generated by $VS.Encode$ (see line 16).

(4) $PT'_i(u) \neq v$ (see line 25).

These four situations in Algorithm 6 will also cause $O'_1(i, u, v, q_2)$ to output *null*: lines 14 - 17 match situation (1); lines 19 - 24 match situations (2) and (3); line 29 matches situation (4).

Case 2: $VS.Encode(i, u, v, q_2)$ does not output *null*. Suppose X is the external input to G' and x^i is the input to PT'_i. For $VS.Encode(i, x^i, PT'_i(x^i), q_2)$, there are three cases (see lines 28 - 33 in Algorithm 2):

(1) If PT'_i's output is an intermediate output and $PT'_i(x^i)$ is an FHE encryption of $(\perp, \perp)$, then $VS.Encode$ decrypts $PT'_i(x^i)$, and outputs $\perp$.

(2) If PT'_i's output is an intermediate output and $PT'_i(x^i)$ is not an FHE encryption of $(\perp, \perp)$, then $VS.Encode$ decrypts $PT'_i(x^i)$, and outputs $\top$.

(3) If PT_i''s output is an external output, then $VS.Encode$ decrypts $PT_i'(x^i)$, and gets $FHE.Dec(PT_i'(x^i))$. $VS.Encode$ outputs the second half of $FHE.Dec(PT_i'(x^i))$.

Similarly, suppose A_2 queries O_1' with $(i, x^i, PT_i'(x^i), q_2)$. Then O_1' calculates $PT_i(u^i)$ where u^i is the corresponding input to PT_i when X is set of external inputs to G (see lines 12 - 32 in Algorithm 6). For $O_1'(i, x^i, PT_i'(x^i), q_2)$, there are three cases (see lines 36 - 38 in Algorithm 6):

(1) If PT_i''s output is an intermediate output and $PT_i(u^i)$ is $(\perp, \perp)$, O_1' outputs $\perp$.

(2) If PT_i''s output is an intermediate output and $PT_i'(x^i)$ is not $(\perp, \perp)$, then O_1' outputs $\top$.

(3) If PT_i''s output is an external output, then O_1' outputs the second half of $PT_i(u^i)$.

Cases 1 and 2 of $VS.Encode(i, x^i, PT_i'(x^i), q_2)$ and $O_1'(i, x^i, PT_i'(x^i), q_2)$ are the same. For Case 3, $VS.Encode$ outputs the second half of $FHE.Dec(PT_i'(x^i))$, while O_1' outputs the second half of $PT_i(u^i)$. According to Lemma 2, $PT_i'(x^i) = FHE.Enc(PT_i(u^i))$, hence $VS.Encode$ and O_1' have the same output.

Therefore, we know that $VS.Encode$ and O_1' have the same input-output functionality.

Now we are going to prove that $VS.Path$ and O_2' have the same input-output functionality.

In steps 1 and 2 of Exp^{real}, A_1 first chooses a table graph G and then $VS.Encrypt$ encrypts G and outputs an encrypted table graph G'. If $PT_{i_1}', ..., PT_{i_p}'$ chosen by A_2 form a path $P = PT_{i_1}' \rightarrow PT_{i_2}' \rightarrow ... \rightarrow PT_{i_p}'$, then $VS.Path(PT_{i_1}', ..., PT_{i_p}')$ returns with an external input X to G' such that the evaluation of G' with X as input includes the evaluation of tables $PT'_{i_1}, ..., PT'_{i_p}$. If $(PT'_{i_1}, ..., PT'_{i_p})$ does not form a path $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$, then $VS.Path$ returns $null$.

Similarly, in step 1 and 2 of Exp, A_1 first chooses a table graph G and then $VS.Encrypt$ encrypts G and outputs an encrypted table graph G'. If $PT'_{i_1}, ..., PT'_{i_p}$ chosen by A_2 form a path $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$, then $O'_2(PT'_{i_1}, ..., PT'_{i_p})$ returns with an external input X to G' such that the evaluation of G' with X as input includes the evaluation of tables $PT'_{i_1}, ..., PT'_{i_p}$. If $(PT'_{i_1}, ..., PT'_{i_p})$ does not form a path $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$, then O'_2 returns $null$.

Thus $VS.Path$ and O'_2 have the same input-output functionality. □

In order to prove Theorem 2, we first prove a lemma which focuses on the security of a single table:

Lemma 6. *For every p.p.t. adversary $A = (A_1, A_2)$, and table $PT_i \in G$, consider the following two experiments:*

$SingleT^{real}(1^K)$:

(1) $(G, C, state_A) \leftarrow A_1(1^K)$. C is of size s and depth d, and the length of the output is m bits. C computes PT_i's rhs function.

(2) $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$

(3) generate a universal circuit $U_{s,d}$ (for circuits of size s and depth d), and a string S_C for C, such that $U_{s,d}(S_C, x) = C_i(x)$.

(4) Let $U = (U_1, ..., U_m)$ where U_i is the circuit that outputs the ith bit of $U_{s,d}$.

(5) $E_C \leftarrow FHE.Encrypt(hpk, S_C)$

```
(6) $a \leftarrow A_2(1^K, (True, E_C), U, hpk, state_A)$
(7) Output $a$
```

```
$SingleT^{ideal}(1^K)$:
(1) $(G, C, state_A) \leftarrow A_1(1^K)$. $C$ is of size $s$ and depth $d$, and the length of the output is $m$ bits. $C$ computes $PT_i$'s rhs function.
(2) $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$
(3) $\tilde{G} \leftarrow S_1(1^K, s, d, G_{struc})$. $\tilde{F_i}$ is the rhs function of $\tilde{PT_i} \in \tilde{G}$.
(4) generate a universal circuit $U_{s,d}$ (for circuits of size $s$ and depth $d$). Convert $\tilde{F_i}$ to a circuit $C'$ such that $C'$ is of size $s$ and depth $d$. Generate a string $S_{C'}$ for $C'$, such that $U(S_{C'}, x) = C'(x)$
(5) Let $U = (U_1, ..., U_m)$ where $U_i$ is the circuit that outputs the $i$th bit of $U_{s,d}$.
(6) $E_{C'} \leftarrow FHE.Encrypt(hpk, S_{C'})$
(7) $a' \leftarrow A_2(1^K, (True, E_{C'}), U, hpk, state_A)$
(8) Output $a'$
```

Then the outputs of the two experiments are computationally indistinguishable.

Proof. A_2's input in $SingleT^{real}$ is $\{1^K, (True, E_C), U, hpk, state_A\}$, while its input in $SingleT^{ideal}$ is $\{(True, E_{C'}), U, hpk, state_A\}$. If A_2's inputs in $SingleT^{real}$ and

$SingleT^{ideal}$ are computationally indistinguishable, then A_2's output a and a' in $SingleT^{real}$ and $SingleT^{ideal}$ are computationally indistinguishable. Because E_C and $E_{C'}$ are two fully homomorphic encrypted ciphertexts, they are computationally indistinguishable under the IND-CPA security of FHE. Hence $\{1^K, (True, E_C), U, hpk, state_A\}$ and $\{1^K, (True, E_{C'}), U, hpk, state_A\}$ are computationally indistinguishable. Therefore a' is computationally indistinguishable from a, which proves the lemma. □

We extend the single table property to the whole system. Assuming there are n tables in $G'' = [TS'', G_{struc}]$, we replace the tables in G'' one by one with the tables in $G' = [TS', G_{struc}]$ and generate a sequence of $n+1$ different table graphs which differ only at one table:

$$\begin{aligned}
TS^0 :& \quad (PT''_1, PT''_2, PT''_3, ..., PT''_n) \\
TS^1 :& \quad (PT'_1, PT''_2, PT''_3 ..., PT''_n) \\
TS^2 :& \quad (PT'_1, PT'_2, PT''_3, ..., PT''_n) \\
\vdots & \qquad\qquad \vdots \\
TS^n :& \quad (PT'_1, PT'_2, PT'_3 ..., PT'_n)
\end{aligned}$$

All the above table sets have the same graph structure G_{struc}. For TS^i, $i \in \{0, , , n\}$, the experiment Exp^i for a p.p.t. adversary pair $A = (A_1, A_2)$ is as follows:

$Exp^i(1^K)$:

1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$

3. generate a universal circuit $U_{s,d}$ (for circuits of size s and depth d)
4. Let $U = (U_1, ..., U_m)$ where U_i is the circuit that outputs the ith bit of $U_{s,d}$.
5. For $j = 1, ..., i$,
   - (a) construct a circuit C_j of size s and depth d which has the same functionality as $PT_j \in G$ and outputs m bits.
   - (b) generate a string S_{C_j} such that $U(S_{C_j}, x) = C_j(x)$.
   - (c) $E_{C_j} \leftarrow FHE.Encrypt(hpk, S_{C_j})$
   - (d) set $(True, E_{C_j})$ as table PT_j^i
6. $\tilde{G} \leftarrow S_1(1^K, s, d, G_{struc})$
7. For $j = i + 1, ..., n$,
   - (a) $\tilde{F_j}$ is the rhs function of $\tilde{PT_j} \in \tilde{G}$. Convert $\tilde{F_j}$ to circuit C'_j. Generate a string $S_{C'_j}$ such that $U(S_{C'_j}, x) = C'_j(x)$.
   - (b) $E_{C'_j} \leftarrow FHE.Encrypt(hpk, S_{C'_j})$
   - (c) set $(True, E_{C'_j})$ as table PT_j^i

   (The tables $(True, E_{C_1}), ..., (True, E_{C_i})$ constructed in step 5 and $(True, E_{C'_{i+1}}), ..., (True, E_{C'_n})$ constructed in step 7 form G^i.)
8. $a \leftarrow A_2^{O'_1, O'_2}(1^K, G^i, G, hpk, CP, U, state_A)$
   O'_1, O'_2 do the following: O'_1 works the same way as $S_2^{O_1}$ (see S_1's construction and Algorithm 6). O'_2 works the same way as $S_3^{O_2}$ (see S_2's construction and Algorithm 7).

```
9. Output $a$
```

Note that Exp^0 is the same experiment as Exp^{ideal}, while Exp^n is the same as Exp in Lemma 5.

The reason that Exp^0 is the same experiment as Exp^{ideal} is as follows. In S_1's construction, S_1 does the same as in Step 6 of Exp^i and generates $\tilde{G}$. Then it runs $VS.Encrypt$ and gets an encrypted version of $\tilde{G}$, which is G''. In Exp^0, Step 5 will not be executed and steps 2,3,4,6,7 do what Algorithm 1 does with $\tilde{G}$ as the algorithm's input. Hence, what steps 2,3,4,6,7 do is running $VS.Encrypt$ to encrypt $\tilde{G}$. Therefore, $G^0 = G''$. Moreover, O'_1 works the same way as $S_2^{O_1}$ and O'_2 works the same way as $S_3^{O_2}$.

The reason that Exp^n is the same experiment as Exp is as follows. In Exp^n, Step 7 will not be executed and steps 2,3,4,5 do what Algorithm 1 does with G as the algorithm's input. Hence, what steps 2,3,4,5 do is running $VS.Encrypt$ to encrypt G. Therefore, $G^n = G'$, where G' is in Exp.

We prove that Exp^{real} is computationally indistinguishable from Exp^{ideal} by contradiction. Assuming that Exp^{real} and Exp^{ideal} are computationally distinguishable, Lemma 5 implies that Exp^0 is computationally distinguishable from Exp. Since Exp is the same experiment as Exp^n, Exp^0 is computationally distinguishable from Exp^n. Namely, there exists a p.p.t. algorithm D such that

$$|Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]| > negl(K)$$

On the other hand,

$$
\begin{aligned}
&|Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]| \\
&\qquad \leq |Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^1(1^K)) = 1]| \\
&\qquad + |Pr[D(Exp^1(1^K)) = 1] - Pr[D(Exp^2(1^K)) = 1]| \\
&\qquad \ldots\ldots \\
&\qquad + |Pr[D(Exp^{n-1}(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]|
\end{aligned}
$$

Hence,

$$
\begin{aligned}
&|Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^1(1^K)) = 1]| \\
&\qquad + |Pr[D(Exp^1(1^K)) = 1] - Pr[D(Exp^2(1^K)) = 1]| + \ldots \\
&\qquad + |Pr[D(Exp^{n-1}(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]| > negl(K)
\end{aligned}
$$

Hence, for at least an $i \in \{0, ..., n-1\}$,

$$|Pr[D(Exp^i(1^K)) = 1] - Pr[D(Exp^{i+1}(1^K)) = 1]| > negl(K)/n = negl(K),$$

where the last equality holds because by the definition of the negligible function $negl(K)$, for all $c > 0$, there exists N such that for all $K > N$, $negl(K) < K^{-c}$. Therefore, for all $c > 0$, there exists N such that for all $K > N$, $negl(K)/n = K^{-c}/n < K^{-c}$, which means $negl(K)/n$ is also a negligible function. Therefore, there are two experiments Exp^i and Exp^{i+1} that are computationally distinguishable. Namely, there exists a pair of p.p.t. adversaries $A = (A_1, A_2)$ and a p.p.t. algorithm

D that can computationally distinguish Exp^i and Exp^{i+1}, i.e.,

$$|Pr[D(Exp^i(1^K)) = 1] - Pr[D(Exp^{i+1}(1^K)) = 1]| > negl(K)$$

We know that in Exp^i and Exp^{i+1} the difference between TS^i and TS^{i+1} is only at the $i+1^{th}$ table. We use $A = (A_1, A_2)$ to construct a pair of p.p.t. adversaries $A' = (A'_1, A'_2)$ with a p.p.t. algorithm D' that contradicts Lemma 6. A' is the pair of adversary in $SingleT^{real}$ or $SingleT^{ideal}$ in Lemma 6 and A' runs the experiments Exp^i and Exp^{i+1} with the pair of p.p.t. adversaries $A = (A_1, A_2)$ to distinguish $SingleT^{real}$ or $SingleT^{ideal}$. A' does the following:

A'_1:

1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. A'_1 constructs a circuit C_{i+1} of size s and depth d which has the same functionality as $PT_{i+1} \in G$ that outputs m bits.
3. output $(G, C_{i+1}, state_{A'})$

A'_2:

0. A'_1 outputs $(G, C_{i+1}, state_{A'})$ as in Step 1 of either $SingleT^{real}$ or $SingleT^{ideal}$ in Lemma 6.
1. A'_2 gets either

$$((True, E_{C_{i+1}}), U, hpk, state_{A'})$$

or

$$((True, E_{C'_{i+1}}), U, hpk, state_{A'})$$

as in step 5 of $SingleT^{real}$ or $SingleT^{ideal}$ in Lemma 6.

2. For $j = 1, ..., i$, A'_2 does the following:
   (a) constructs a circuit C_j of size s and depth d which has the same functionality as $PT_j \in G$ that outputs m bits. (G comes from Line 1 of A'_1)
   (b) generates a string S_{C_j} for C_j such that $U_{s,d}(S_{C_j}, x) = C_j(x)$.
   (c) $E_{C_j} \leftarrow FHE.Enc(hpk, S_{C_j})$
   (d) constructs $(True, E_{C_j})$
3. constructs $\tilde{G} = [\tilde{TS}, G_{struc}]$ the same way as S_1 would construct. $\forall \tilde{PT}_i \in \tilde{TS}$, $\tilde{PT}_i = (True, \tilde{F}_i)$, where $\tilde{F}_i$ is a function that can be converted into a size s, depth d circuit that outputs m bits.
4. For $j = i + 2, ..., n$, A'_2 does the following:
   (a) Converts $\tilde{F}_j$ to circuit C'_j. Generates a string $S_{C'_j}$ such that $U(S_{C'_j}, x) = C'_j(x)$.
   (b) $E_{C'_j} \leftarrow FHE.Enc(hpk, S_{C'_j})$
   (c) constructs $(True, E_{C'_j})$

(The tables $(True, E_{C_1}), ..., (True, E_{C_i})$ constructed in step 2 and $(True, E_{C'_{i+2}}), ..., (True, E_{C'_n})$ constructed in step 3 with the table A'_2 gets in step 1 (which is either $(True, E_{C_{i+1}})$ or $(True, E_{C'_{i+1}})$) will form either G^i or G^{i+1} in Exp^i or Exp^{i+1}.)

5. A'_2 runs A_2 as follows:
$a \leftarrow A_2^{OA_1,OA_2}(G^i, G, hpk, UC, state_A)$ or
$a \leftarrow A_2^{OA_1,OA_2}(G^{i+1}, G, hpk, UC, state_A)$
(A_2's oracle queries are answered by OA_1 and OA_2, which are constructed by A'_2 as follows: OA_1 is the same as $S_2^{O_1}$. OA_2 is the same as $S_3^{O_2}$. Moreover, OA_1 and OA_2 know the public key hpk and the table graph G^i or G^{i+1} A'_2 constructs at step 4. Hence OA_1, OA_2 play the role of O'_1, O'_2 in experiments Exp^i and Exp^{i+1}). Namely,

$$a \leftarrow A_2^{O'_1,O'_2}(G^i, G, hpk, U, state_A)$$

or

$$a \leftarrow A_2^{O'_1,O'_2}(G^{i+1}, G, hpk, U, state_A)$$

If in step 1 of A'_2, A'_2 gets

$$((True, E_{C_{i+1}}), U, hpk, state_{A'})$$

from $SingleT^{real}$, then after step 4, A'_2 constructs G^{i+1}, and A'_2 is essentially simulating Exp^{i+1}.

If in step 1 of A'_2, A'_2 gets

$$((True, E_{C'_{i+1}}), U, hpk, state_{A'})$$

from $SingleT^{ideal}$, then after step 4, A'_2 constructs G^i, and A'_2 is essentially simulating Exp^i.

Hence, we know from the above construction that

$$Pr[D'(SingleT^{real}(1^K)) = 1] = Pr[D(Exp^{i+1}(1^K)) = 1],$$
$$Pr[D'(SingleT^{ideal}(1^K)) = 1] = Pr[D(Exp^{i}(1^K)) = 1].$$

which implies that

$$|Pr[D'(SingleT^{real}(1^K)) = 1] - Pr[D'(SingleT^{ideal}(1^K)) = 1] > negl(K)$$

Consequently A' contradicts Lemma 6. and Theorem 2 is proved.

□

Theorem 3. *The verification scheme VS in Subsection 3.2.3 satisfies Definition 16.*

Proof. Theorems 1 and 2 imply Theorem 3. □

Algorithm 2 VS.Encode(i, u, v, q)

```
1:  if $q$ is of the kind $q_1$ then
2:      if $|u| \neq m$ or the first half of $u$ is not $\top$ then /* $m$ is defined in line 3 in Algorithm 1*/
3:          return $null$
4:      else
5:          $w \leftarrow FHE.Enc(hpk, u)$
6:          add $(i, w, null)$ to $S$
7:      end if
8:      return $u$
9:  end if
10: if $q$ is of the kind $q_2$ then
11:     for all $x^i_j \in u$ do
12:         if according to $G'$, $x^i_j$ should be an output of $PT'_k$ then
13:             if $\nexists(k, x^k, PT'_k(x^k)) \in S$ such that $x^i_j = PT'_k(x^k)$ then return $null$
14:             else if $\exists(k, x^k, PT'_k(x^k)) \in S$ such that $x^i_j = PT'_k(x^k)$ and $PT'_k(x^k)$ is an FHE encryption of $(\perp, \perp)$ then return $null$
15:             end if
16:         else if according to $G'$, $x^i_j$ should be an external input then
17:             if $\nexists(i, w, null) \in S$ such that $x^i_j = w$ then return $null$
18:             end if
19:         end if
20:     end for
21:     for all $k \in \{1, ..., m\}$ do
22:         $s_k = FHE.Eval(hpk, U_k, u, E_{C_i})$
23:     end for
24:     $s \leftarrow s_1 s_2 ... s_m$
25:     if $s \neq v$ then return $null$
26:     else
27:         add $(i, u, v)$ to $S$
28:         for all $i \in \{1, ..., m\}$ do
29:             $b_i \leftarrow FHE.Dec(hsk, s_i)$
30:         end for
31:         if $v$ is not an external output and $b_1 b_2 ... b_{m/2} \neq \perp$ then return $\top$
32:         else return $b_{m/2+1} b_2 ... b_m$
33:         end if
34:     end if
35: end if
```

Algorithm 3 VS.Eval(1^K,QA_E,G',hpk,U)

```
1: for all $(Q_i, A_i) \in QA_E$ do
2:     if $Q_i$ is $(r, u, v, q_2)$ then
3:         $PT'_r(u) \leftarrow FHE.Eval(hpk, U, E_{C_r}, u)$ /* $PT'_r = (True, E_{C_r})$*/
4:         if $PT'_r(u) \neq v$ then
5:             return 0
6:         end if
7:     end if
8: end for
9: runs $V'(1^K, G', F_{spec}, VGA_r, CP)$ as follows.
10: for all table $PT'_i(x^i)$ $V'$ chooses to evaluate do
11:     if $\nexists(Q_j, A_j) \in QA_E$ where $Q_j = (i, x^i, PT'_i(x^i), q_2)$ then
12:         return 0
13:     end if
14: end for
15: if $V'$'s output is not equal to $V$'s output then
16:     return 0
17: end if
18: return 1
```

Algorithm 4 VS.Path($PT'_{i_1}, ..., PT'_{i_p}$)

```
1: if $(PT'_{i_1}, ..., PT'_{i_p})$ form a path P such that $P = PT'_{i_1} \rightarrow PT'_{i_2} \rightarrow ... \rightarrow PT'_{i_p}$ then
2:     generate an external input X to the table graph G such that the evaluation of
   G includes the evaluation of tables $PT_{i_1}, ..., PT_{i_p}$.
3:     return X
4: else return $null$
5: end if
```

Algorithm 5 Evaluation of G' with an external input X

```
1: For $PT'_i$ chosen by $V$ in the consistent manner /* see Definition 5*/
2: for all $x^i_j \in x^i$ do
3:     if $x^i_j$ is an external input that is not null then
4:         the verifier asks the developer to call $VS.Encode(i, x^i_j, null, q_1)$
5:         $Enc_{x^i_j} \leftarrow VS.Encode(i, x^i_j, null, q_1)$. $x^i_j \leftarrow Enc_{x^i_j}$
6:     else if $x^i_j$ is an external input that is null then
7:         the verifier skips evaluating $PT'_i$ and regards the output of $PT'_i$ as null
8:         break
9:     else if $x^i_j$ is an intermediate input that corresponds to an incoming edge from
   another table $PT'_p$ then
10:        if $PT'_p$'s output is null then
11:            the verifier skips evaluating $PT'_i$ and regards the output of $PT'_i$ as null
12:            break
13:        else if $VS.Encode(p, x^p, PT'_p(x^p))$ is $\perp$ then
14:            the verifier skips evaluating $PT'_i$ and regards the output of $PT'_i$ as null
15:            break
16:        else $x^i_j \leftarrow PT'_p(x^p)$
17:        end if
18:    end if
19: end for
20: if the evaluation of $PT'_i$ is skipped then continue
21: else
22:     The verifier evaluates $PT'_i(x^i)$ /* see Remark 7*/
23:     The verifier then sends $(i, x^i, PT'_i(x^i), q_2)$ to the developer
24:     The developer returns $VS.Encode(i, x^i, PT'_i(x^i), q_2)$ to the verifier
25: end if
26: Find the tables $PT'_{i_1}, ..., PT'_{i_s}$ that have external outputs according to $G'$ Let
    $Y = \{y_{i_1}, ..., y_{i_s}\}$ be their outputs
27: return $Y$
```

Algorithm 6 $O_1(i, u, v, q)[id, S, H]$

```
if  $q$ is of the kind $q_1$ then
    if $|u| \neq m$ or the first half of $u$ is not $\top$ then /* $m$ is defined in line 3 in Algorithm 1*/
        return $null$
    else
        $p \leftarrow FHE.Enc(hpk, u)$
        add $(i, p, null, id)$ to $S$
        add $(i, u, null, id)$ to $H$
        $id \leftarrow id + 1$
        return $p$
    end if
end if
if $q$ is of the kind $q_2$  then
    for all $x^i_j \in u$ do
        if according to $G''$, $x^i_j$ should be an external input then
            if $\exists(i, x^i_j, null, id) \in S$ then
                get $(i, x^i_j, null, id) \in H$ and let $e^i_j \leftarrow x^i_j$
            else return $null$
            end if
        else if according to $G''$, $x^i_j$ should be an output of $PT'_k$ then
            if $\exists(k, x^k, PT''_k(x^k), id) \in S$ where $x^i_j = PT''_k(x^k)$ then
                get $(k, x^k, PT_k(x^k), id) \in H$ and let $e^i_j \leftarrow PT_k(x^k)$
                if $PT_k(x^k) = (\perp, \perp)$ then  return $null$
                end if
            else return $null$
            end if
        end if
    end for
    $s = FHE.Eval(hpk, U, u, E_{C'_i})$
    if $s \neq v$ then return $null$
    else
        $e^i \leftarrow e^i_1 e^i_2 ... e^i_m$
        add $(i, e^i, PT_i(e^i), id)$ to $H$
        add $(i, u, v, id)$ to $S$
        $id \leftarrow id + 1$
    end if
    if $PT_i(e^i) \neq (\perp, \perp)$ and is not an external output then return $\top$
    else return the second half of $PT_i(e^i)$
    end if
end if
```

Algorithm 7 $O_2(PT''_{i_1}, ..., PT''_{i_p})$

```
if $PT''_{i_1}, ..., PT''_{i_p}$ form a path P such that $P = PT''_{i_1} \rightarrow PT''_{i_2} \rightarrow ... \rightarrow PT''_{i_p}$ then
    generate an external input X to the table graph G such that the evaluation of G with X as input includes the evaluation of tables $PT_{i_1}, ..., PT_{i_p}$. The method to generate this X is the same as the method used in Algorithm 4.
    return X
else return null
end if
```

Chapter 4

Secure and Trusted Verification for Malicious Developers

The structure of this chapter is similar to Chapter 3. In Section 4.1 we will discuss the general case of secure and trusted verification in the context of a malicious developer, and in Section 4.2, we will discuss our implementation of secure and trusted verification.

4.1 Secure and trusted verification for malicious developers

In Chapter 3 we discussed what a general secure and trusted verification scheme should be like in the context of an honest developer, and also gave the definition of an honest developer in Definition 19. But we did not give a definition of a malicious developer, or what potential problems a malicious developer could bring to the scheme. We are going to discuss them in this section.

In Figure 3.1 the verifier asks the developer to run $VS.Encode$. Thus, if the developer is malicious, then the developer can actually replace $VS.Encode$ with some other malicious algorithm $VS.Encode'$ and return an output of $VS.Encode'$. If we do not provide a method to prevent this scenario from happening, then a buggy implementation could pass the verifier's verification when it actually should not.

Bearing this in mind, we define a new verification scheme VS below in Definition 20, based on the verification scheme in Definition 16, by adding an algorithm $VS.Checker$ to prevent the developer from being malicious. In the definition, $VS.Encrypt$ and $VS.Encode$ remain the same. In the context of a malicious developer, the developer may call some other algorithm $VS.Encode'$ instead of $VS.Encode$. By asking the developer to run $VS.Checker$, the verifier can determine whether the developer actually runs $VS.Encode$. Moreover, because $VS.Checker$ itself is also run by the developer, it can also be replaced by some other algorithm $VS.Checker'$. As a result, $VS.Checker$ is designed such that even if it is replaced with some other algorithm, the verifier can still figure out that the answers from the developer by running $VS.Checker'$ or $VS.Encode'$ are not honest replies.

$VS.Eval$ is an algorithm similar to the $VS.Eval$ in Definition 16. By running $VS.Eval$ with a publicly known $Certificate$, any third party can check whether the developer is malicious and whether the verifier does the correct verification. Figure 4.1 is the description of the protocol between the developer and the verifier.

Definition 20 (Extension of definition 16).
A secure and trusted verification scheme VS is a tuple of p.p.t. algorithms ($VS.Encrypt$, $VS.Encode, VS.Checker, VS.Eval$) *such that*

(1) VS.Encrypt is a p.p.t. algorithm that takes the security parameter 1^K and the table graph G as inputs and outputs an encrypted table graph G'.

(2) VS.Encode is a p.p.t. algorithm that takes an input x and returns an encoding Enc_x.

(3) VS.Checker is a p.p.t. algorithm with a memory $state_C$ that takes as input Q and outputs A.

(4) VS.Eval is a p.p.t. algorithm that takes as input $(1^k, Certificate)$ and outputs 1 or 0, where $Certificate$ is a piece of public information. VS.Eval has a verifier V' hardcoded in it. V' is an honest verifier that satisfies Definition 7.

VS satisfies Definition 21 and 22 below.

Definition 21 (Correctness)**.**
If $VS.Eval(1^k, Certificate) = 1$, then the following must be satisfied:

(1)

$$Pr_r[V(1^K, G', F_{spec}, VGA_r, CP)(r) = V'(1^K, G', F_{spec}, VGA_r, CP))(r)] \geq 1 - negl(K), \quad (4.1)$$

where the probabilities are over the random bits r of V.

(2) For any input x_1 of VS.Encode and any input x_2 of VS.Checker,

$$Pr[VS.Encode(x_1) = VS.Encode'(x_1), VS.Checker(x_2) = VS.Checker'(x_2)] \geq 1 - negl(K) \quad (4.2)$$

(3)

$$Pr_r[V(1^K, G, F_{spec}, VGA_r, CP)(r) = V(1^K, G', F_{spec}, VGA_r, CP)(r)] \geq 1 - negl(K), \quad (4.3)$$

Definition 22 (Security)**.** *For $A = (A_1, A_2)$ and $S = (S_1, S_2, S_3)$ which are p.p.t algorithms, let*

1. *SI consist of any piece of information that the scheme VS wants to hide and all the information that can be known from it.*
2. *AI is the input of the oracle O_2 in Exp^{ideal} below. AI consist of any piece of secret information from A.*
3. *The oracle O_1 in Exp^{ideal} below output $O_1(x) \notin SI$ for any input x that S_2 chooses.*
4. *The oracle O_2 in Exp^{ideal} below output $O_2(x) \notin SI$ for any input x that S_2 chooses.*

Consider the following two experiments,

```
$Exp^{real}(1^K)$:
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', CP') \leftarrow VS.Encrypt(1^K, G, CP)$
3. $a \leftarrow A_2^{VS.Encode, VS.Checker}(1^K, VS.Eval, G', G, CP, CP', state_A)$
4. Output $a$
```

```
$Exp^{ideal}(1^K)$:

1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G'', CP'') \leftarrow S_1(1^K)$
3. $a \leftarrow A_2^{S_2^{O_1}, S_3^{O_2(AI)}}(1^K, VS.Eval, G'', G, CP, CP'', state_A)$
4. Output $a$
```

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

$$|Pr[D(Exp^{ideal}(1^K)_{K\in\mathbb{N}}, 1^K) = 1] - Pr[D(Exp^{real}(1^K)_{K\in\mathbb{N}}, 1^K) = 1]| \leq negl(K)$$

Figure 4.1 below describes the protocol between the developer and the verifier when the scheme VS is applied.

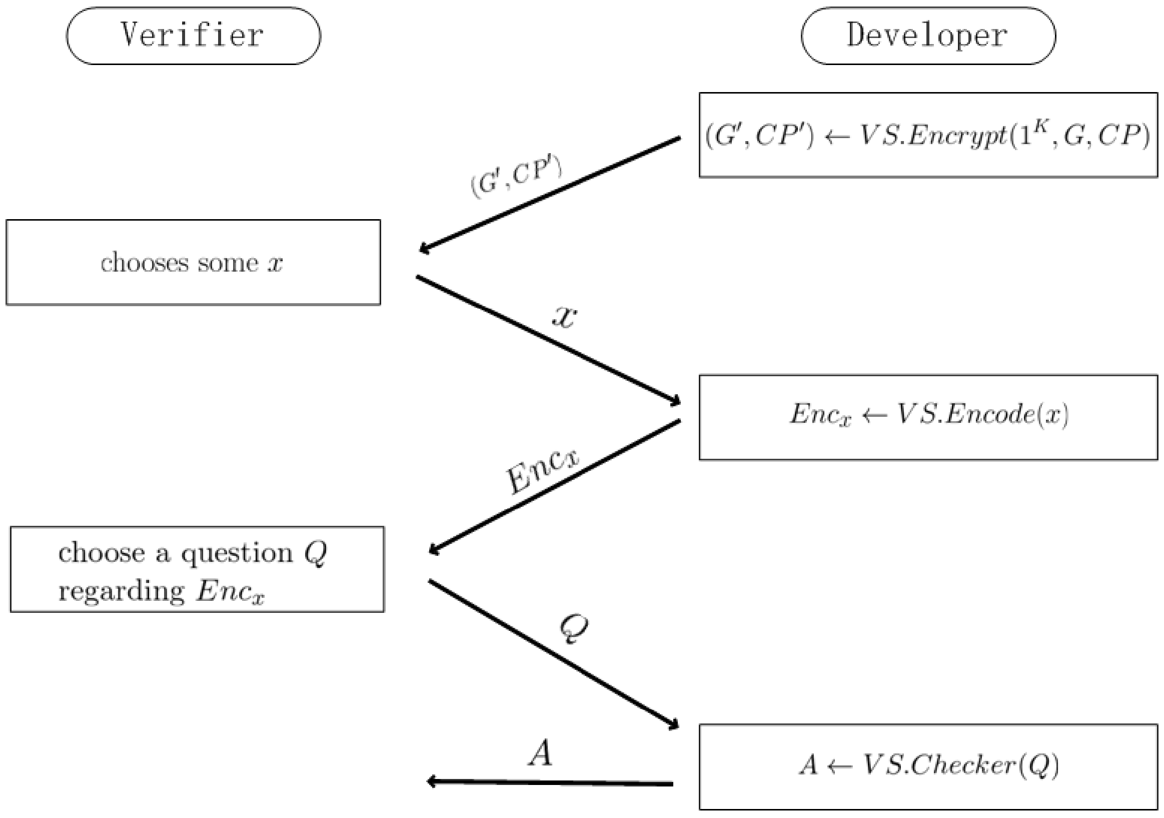

Verifier
Developer
$(G', CP') \leftarrow VS.Encrypt(1^K, G, CP)$
(G', CP')
chooses some x
x
$Enc_x \leftarrow VS.Encode(x)$
Enc_x
choose a question Q regarding Enc_x
Q
$A \leftarrow VS.Checker(Q)$
A

Figure 4.1: The protocol of VS in Definition 20

Definition 23. *A* malicious developer *is a developer that in Figure 4.1 runs $VS.Encode'$ with the verifier's question x and $VS.Checker'$ with the verifier's question Q, so that at least one of the following is true:*

1. *$VS.Encode'$ does not have the same input-output functionality as $VS.Encode$.*
2. *$VS.Checker'$ does not have the same input-output functionality as $VS.Checker$.*

Definition 24. *An* honest *developer in this chapter is a developer that runs $VS.Encode$ with the verifier's question x and runs $VS.Checker$ with the verifier's question Q in Figure 4.1.*

Remark 9. *In Step 3 of Exp^{real} in Definition 22, $VS.Encode$ and $VS.Checker$ are not oracles of A_2. A_2 plays the role of a malicious verifier and A_2 interacts with the developer just like in Figure 4.1. A_2 asks the developer to run $VS.Encode$ or $VS.Checker$ with an input it chooses. Similarly, in Step 3 of Exp^{ideal} in Definition 22, $S_2^{O_1}$ and $S_3^{O_2}$ are also not oracles of A_2. A_2 plays the role of a malicious verifier and A_2 interacts with the developer, who unlike in Exp^{real}, runs $S_2^{O_1}$ instead of $VS.Encode$ and $S_3^{O_2}$ instead of $VS.Checker$. Whenever we say A_2 queries $VS.Encode$ or $VS.Checker$, we mean A_2 asks the developer to run $VS.Encode$ or $VS.Checker$. Similarly, whenever we say A_2 queries $S_2^{O_1}$ or $S_3^{O_2}$, we mean A_2 asks the developer to run $S_2^{O_1}$ or $S_3^{O_2}$.*

Remark 10. *The developer in Exp^{real} and Exp^{ideal} of Definition 18 is an honest developer.*

4.2 Our implementation with malicious developers

4.2.1 Technique outline

Our implementation in this section is an extension of our implementation in Section 3.2. The non-disclosed information of both schemes is the same. $VS.Encrypt$, $VS.Encode$ and $VS.Path$ are the same as $VS.Encrypt$, $VS.Encode$ and $VS.Path$ in Section 3.2.3. The difference is that our implementation in this section is in the context of a malicious developer. As mentioned in Section 4.1, VS needs to provide an algorithm $VS.Checker$ to allow the verifier to check whether the replies generated from the developer by running $VS.Encode'$ are the honest replies which $VS.Encode$ would output.

VS.Checker

We use Figure 4.2 to illustrate the general idea of implementing $VS.Checker$. Figure 4.2 is an example of the evaluation of a table graph G on the left and its encrypted version G' on the right. At this point, no $VS.Checker$ is added to the scheme, and we explain below what kind of $VS.Checker$ we need in order to prevent the developer from being malicious.

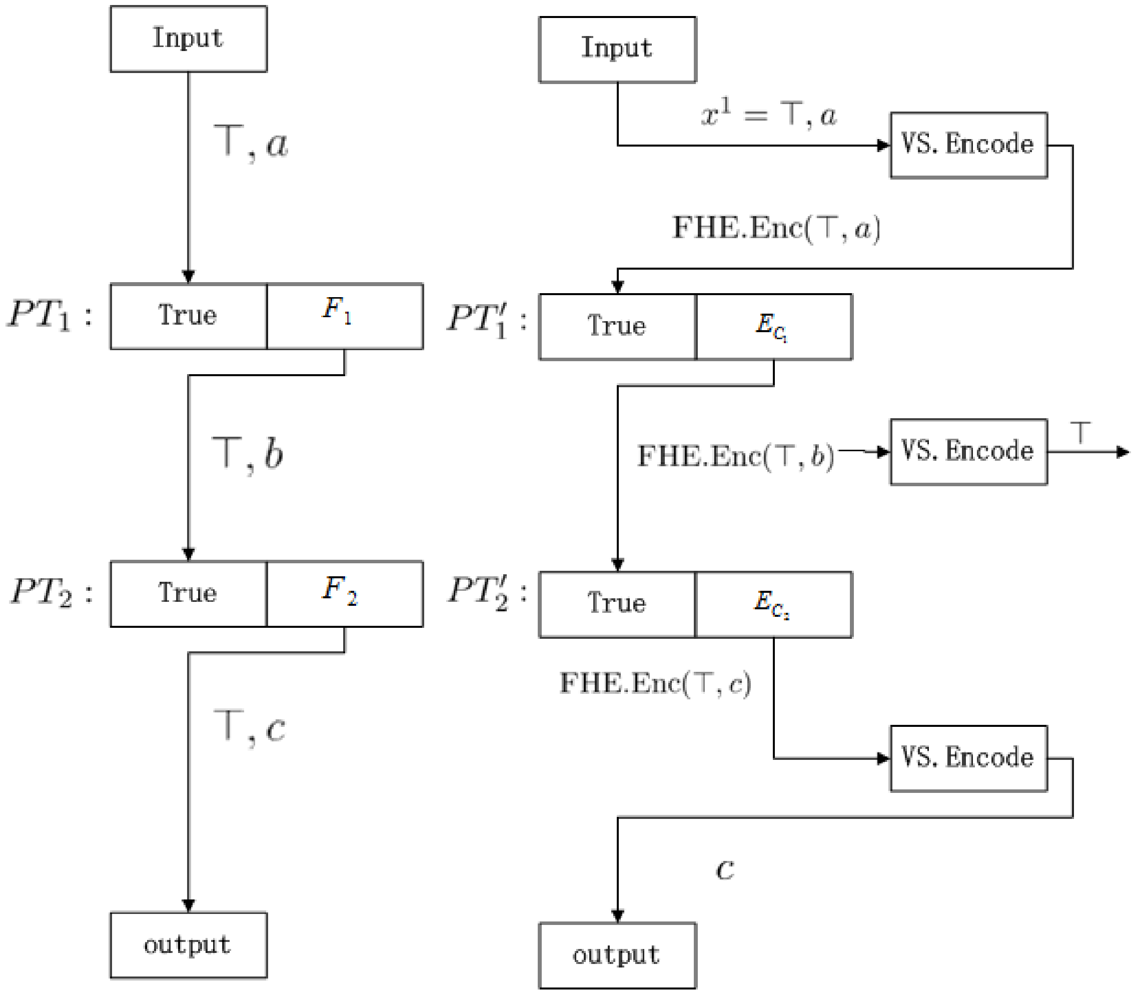


Figure 4.2: An example of the evaluation of table graph G and G'

According to $VS.Encode$'s construction in Algorithm 2, there are three cases which we are going to study below.

Case 1: First, suppose V evaluates a table whose output is an external output. Then V asks the developer to run $VS.Encode$ to decrypt the external output. In Figure 4.2's rhs table graph G', the external output of table PT_2' is $FHE.Enc(\top, c)$, and the correct output of $VS.Encode$ regarding $FHE.Enc(\top, c)$ should be c. However, if the developer is malicious, it can replace $VS.Encode$ with $VS.Encode'$ which outputs c'. This c' can be a corrected output according to F_{spec}, while c is a wrong output that does not match the expected output of F_{spec}. (Given F_{spec} and the external input X to G', this is easy to know.) Consequently, the developer can deceive the verifier into believing that the table graph is bug-free when, in fact, it has a bug (the output c is not a correct output).

For the output $FHE.Enc(\top, c)$ of table PT_2' in Figure 4.2, V wants to know the value c. But it wants to make sure that the developer returns c, and not another value c'. V can use the following method:

Step 1 V chooses a secret key sk of a deterministic private key encryption scheme SE (see Definition 8) predetermined by the verification scheme VS. (SE's encryption algorithm is $SE.Enc$.)

Step 2 V first extracts the second half of $FHE.Enc(\top, c)$, which is $FHE.Enc(c)$. Then V runs $FHE.Eval$:

$$y \leftarrow FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(c))$$

Step 3 V sends y to the developer and asks it to run $VS.Checker$ ($VS.Checker$'s job is to fully homomorphically decrypt y), and the developer returns $VS.Checker$'s output $FHE.Dec(y)$, which we denote as d.

Step 4 V uses SE's decryption algorithm $SE.Dec$ to decrypt d and gets $SE.Dec(sk, d)$.

Step 5 V compares $SE.Dec(sk, d)$ with c. If they are the same, we know that the developer indeed runs $VS.Encode$, which means that the developer is honest, otherwise we know the developer is malicious.

According to Definition 11, $y = FHE.Enc(SE.Enc(sk, c))$. Then, if V sends y to the developer and asks it to run $VS.Checker$, the developer has no choice but to honestly run $VS.Checker$ with y as input. The reason is as follows. First assume that the developer evaluates $VS.Checker(y)$. Then $VS.Checker$ outputs $FHE.Dec(y)$ and the developer returns this output to V. As mentioned above, $y = FHE.Enc(SE.Enc(sk, c))$, and therefore $d = FHE.Dec(y) = SE.Enc(sk, c)$. After V gets d, V obtains c by evaluating $SE.Dec(sk, d)$. Then V compares c with the answer from the developer who is expected to run $VS.Encode$. If they are the same, we know that the developer indeed runs $VS.Encode$, which means that the developer is honest, otherwise we know that the developer is malicious.

Now suppose that the developer runs some other algorithm $VS.Checker'$ instead of $VS.Checker$ and replies V with another value d'. Then V uses $SE.Dec$ with the secret key sk to decrypt d' and gets $SE.Dec(sk, d')$ (which is not c). Also we assume that the developer runs $VS.Encode'$ and returns a c'. If $VS.Checker'$ can generate $SE.Enc(sk, c')$, then after the developer returns $d' = SE.Enc(sk, c')$ to V, V will decrypt $SE.Enc(sk, c')$ and get c', which is also the output of $VS.Encode'$. If this is the case, then V will not be able to know whether the developer is malicious or not. However, we need to point out that $SE.Dec(sk, d')$ has little chance to be c', because $VS.Checker'$ outputs d' without knowing the secret key sk of the private key encryption scheme SE. So even if $VS.Checker'$ intends to output $SE.Enc(sk, c')$, it has a very small probability to actually generate $SE.Enc(sk, c')$. Hence, with a small negligible error, by comparing c' and $SE.Dec(sk, d')$, V can find out that $VS.Encode'$'s output is not correct, even if the developer runs $VS.Checker'$ instead of $VS.Checker$.

Case 2: Now consider the case in Figure 4.2, where V gets the intermediate output $FHE.Enc(\top, b)$ and asks the developer to run $VS.Encode$ with a question regarding this intermediate output. This is different from Case 1 because $FHE.Enc(\top, b)$ is an intermediate output. What V is allowed to know from the developer is whether this intermediate output is meaningful or not. Namely, whether $FHE.Enc(\top, b)$ contains $\top$ or $\perp$. (In Section 2.2.2 we have explained the transformation of an initial table graph to a table graph G in our construction. If a table PT_i in G outputs a symbol $\perp$, then it means that the corresponding row's predicate in the initial table graph is not satisfied. Thus PT_i's output is not "meaningful". That is why V needs to know whether the output of the encrypted version of PT_i is meaningful or not.) A malicious developer can run $VS.Encode'$ instead of $VS.Encode$ and return $\perp$ for the query regarding $FHE.Enc(\top, b)$ instead of $\top$.

For this case V can use Case 1 with the following change: In Step 2, V first extracts the first half of $FHE.Enc(\top, c)$, which is $FHE.Enc(\top)$. Then V runs $FHE.Eval$

$$y \leftarrow FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(\top))$$

Here we explain why we require the tables in standard table graph G in Section 2.2.2 to output values in the format $(flag, x)$, where $flag = \top$ or $\perp$ is a symbol to indicate whether x is a meaningful or meaningless output (denoted as $\perp$). We still use the example in Figure 4.2. Suppose $(True, F_1)$ outputs only b instead of $(\top, b)$ and correspondingly PT_1' outputs only $FHE.Enc(b)$ instead of $FHE.Enc((\top, b))$. Then we are not able to apply the method in Case 1 to this intermediate output. The reason is as follows. If V only gets $FHE.Enc(b)$ as the intermediate output, then in Step 2 of Case 1, V runs $FHE.Eval$ to get:

$$y \leftarrow FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(b))$$

Then in Step 3 of Case 1, V sends y to the developer and asks the developer to run $VS.Checker$. $VS.Checker$ outputs $FHE.Dec(y)$ which in Step 4 of Case 1, will be decrypted by V using $SE.Dec$, and as a result V will know the value of b. The reason that V can obtain b can be explained by the following equations:

$$\begin{gathered} y = FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(b) \\ = FHE.Enc(hpk, SE.Enc(sk, b)) \\ SE.Dec(FHE.Dec(y)) = SE.Dec(SE.Enc(sk, b)) = b \end{gathered}$$

However, for an intermediate output, b should not be revealed to V (this value is secret). Therefore, by expanding the output of PT_1 from b to $(\top, b)$ and correspondingly expanding the output of PT_1' from $FHE.Enc(b)$ to $FHE.Enc(\top, b)$, we can apply the method in Case 1 to this case as well.

Case 3: In this case, V sends a question regarding the external input $(\top, a)$ of table PT_1' in Figure 4.2 to the developer. The developer runs $VS.Encode$, which returns $FHE.Enc(\top, a)$. Here the verifier can also use Case 1 (asking the developer to run $VS.Checker$) to confirm that the answer $FHE.Enc(\top, a)$ is actually a fully homomorphic encryption of $(\top, a)$.

The method in Case 1 does prevent the developer from deceiving V, but by allowing V to ask the developer to run $VS.Checker$ actually makes V far more powerful. There is a risk that V abuses this power to do something malicious. For example, the verifier can use this method to decrypt intermediate outputs, like $FHE.Enc(\top, b)$ in Figure 4.2. What's more, V can even figure out the table contents (e.g., C_1, C_2 and consequently F_1, F_2) by hacking the method as follows. V generates fully homomorphically encrypted ciphertexts maliciously (e.g., extracting certain bits from encrypted circuits such as E_{C_1} and E_{C_2}) and interact with the developer as described in the method of Case 1 to get the secret plaintexts.

Thus we must restrict what kind of question V can ask. For example, in Case 1 Step 3, when V sends y to the developer and asks it to run $VS.Checker$, $VS.Checker$ must include a procedure to check whether $y = FHE.Enc(SE.Enc(sk, c))$ before it decrypts y. But in order to do that, $VS.Checker$ needs to know the secret key sk, which makes the method in Case 1 pointless.

Our solution to this problem is to use bit commitment protocols during the interaction between the verifier V and the developer. The idea of a bit commitment protocol is similar to hiding information in an envelope (The analogy between the envelope and bit commitment protocols is from (Kilian, 1989)). The developer runs $VS.Checker$ which is doing the following: First, given an input $Q_i = (i, p, y)$, $VS.Checker$ fully homomorphically decrypts y and gets a value d. The developer puts d in a sealed envelope and gives it to V. V cannot open the envelope at this stage. After that, $VS.Checker$ needs V to provide a proof that y is indeed $FHE.Enc(SE.Enc(sk, c))$. The proof contains the secret key sk, but revealing sk to $VS.Checker$ does not matter, because $VS.Checker$'s output is already given to V and cannot change at this point. If $VS.Checker$ confirms that y is indeed $FHE.Enc(SE.Enc(sk, c))$, then it generates a key to the sealed envelope and the developer will give this key to V. Then V can know d, using the key to open the envelope. If $VS.Checker$ figures out that y is not $FHE.Enc(SE.Enc(sk, c))$, it simply refuses to generate the key to the envelope. Such an envelope solves the problem of mutual distrust and potential malicious activities by both entities. A bit commitment protocol can play the role of such an envelope.

VS.Eval

$VS.Eval$ is an extension of $VS.Eval$ in Section 3.2.3. It not only needs to check whether the verifier V evaluates the table graph correctly, but it also needs to check whether the developer replies honestly. In order to check whether the verifier V evaluates the table graph correctly, $VS.Eval$ in Algorithm 9 includes Algorithm 3.

On the other hand, in order to check whether the developer replies honestly, $VS.Eval$ reads a log file which records the interaction between the verifier and the developer. By reading this log file, $VS.Eval$ can check whether the verifier actually asks the developer to run $VS.Checker$ for every answer it gets from $VS.Encode$.

Moreover, by reading this log file, VS.Eval can check whether the developer is malicious. Actually, $VS.Eval$ can repeat the whole interaction between the verifier and the developer from the log file.

4.2.2 Construction

We give our implementation of $VS = (VS.Encrypt, VS.Encode, VS.Checker, VS.Eval, VS.Path)$ below. $VS.Encrypt$, $VS.Encode$ and $VS.Path$ are the same as $VS.Encrypt$,

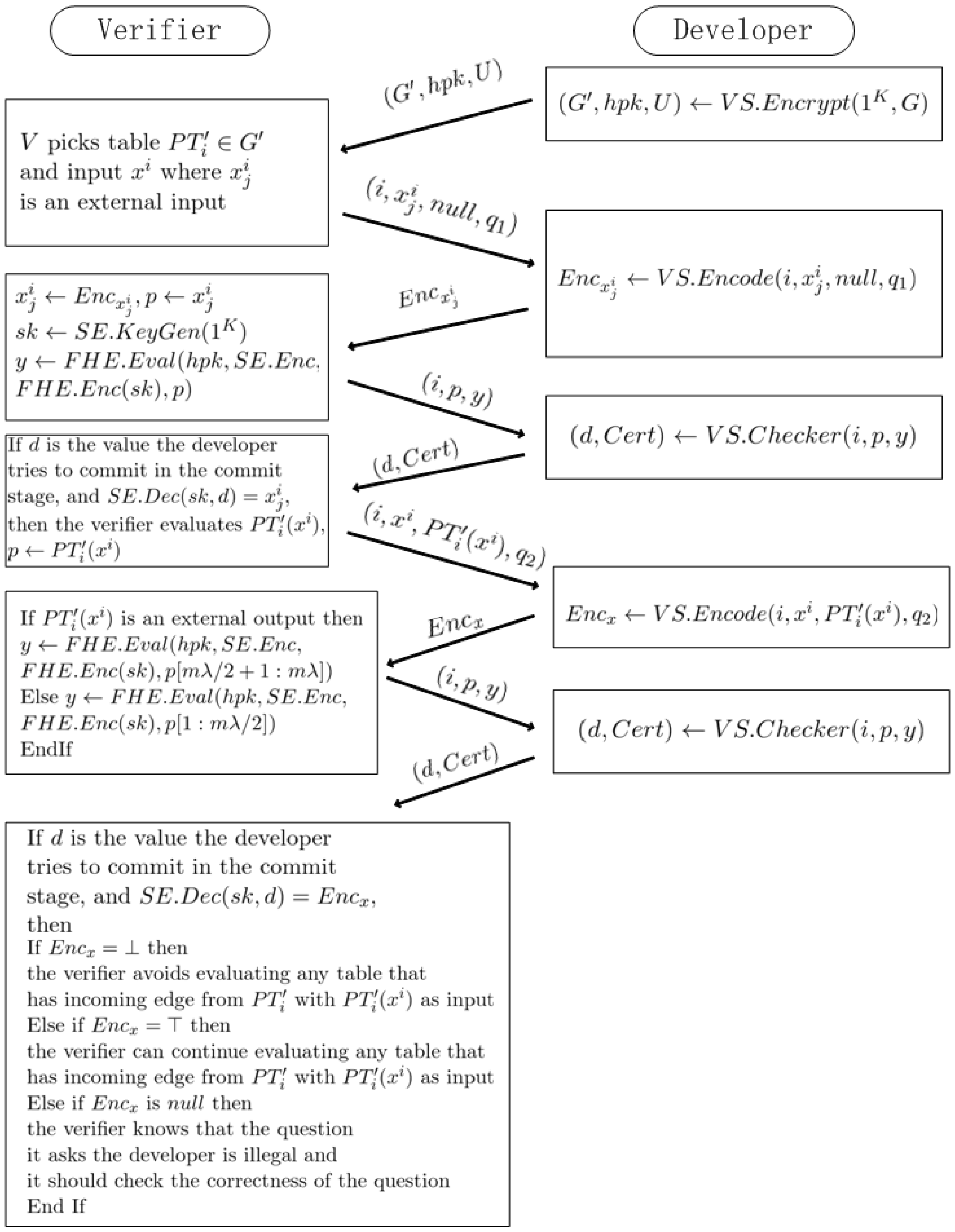
$VS.Encode$ and $VS.Path$ in Section 3.2.3. QA_E is the same as in Section 3.2.3. $(Q_i, A_i, S_i) \in QA_C$, is a tuple of the question Q_i asked by V, the information S_i generated by both the developer and the verifier during the bit commitment protocol and the answer A_i returned by the developer by running $VS.Checker$.

The relation of QA_E and QA_C is as follows: For any $(Qe_i, Ae_i) \in QA_E$, the verifier will generate a corresponding question Qc_i (see Figure 4.3) and send this question to the developer. The developer, after running $VS.Checker$, will return an answer Ac_i.

$VS.Checker$ (see Algorithm 8) takes an input $Q_i = (i, p, y)$ from the verifier about a particular table PT'_i, and the developer launches a bit commitment protocol with the verifier during running $VS.Checker$. The information exchange during the protocol is stored in S_i. In the end it outputs an answer A_i. The F_{secret} in Algorithm 8 contains $FHE.Enc(sk)$ and $SE.Enc$, where sk and $SE.Enc$ are the secret key sk and the encryption algorithm $SE.Enc$ mentioned in Case 1 of the $VS.Checker$ subsection in Section 4.2.1.

$VS.Eval$ (see Algorithm 9) takes as input $(1^K, QA_E, QA_C, G', hpk, U)$ where G', hpk, U are from the output of $VS.Encrypt$.

The protocol of VS (see Figure 4.3) describes what the developer and the verifier should do in VS.



$(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$

V picks table $PT_i' \in G'$ and input x^i where x_j^i is an external input

$Enc_{x_j^i} \leftarrow VS.Encode(i, x_j^i, null, q_1)$

$x_j^i \leftarrow Enc_{x_j^i}, p \leftarrow x_j^i$
$sk \leftarrow SE.KeyGen(1^K)$
$y \leftarrow FHE.Eval(hpk, SE.Enc,$
$FHE.Enc(sk), p)$

$(d, Cert) \leftarrow VS.Checker(i, p, y)$

If d is the value the developer tries to commit in the commit stage, and $SE.Dec(sk, d) = x_j^i$, then the verifier evaluates $PT_i'(x^i)$, $p \leftarrow PT_i'(x^i)$

$Enc_x \leftarrow VS.Encode(i, x^i, PT_i'(x^i), q_2)$

```
If $PT_i'(x^i)$ is an external output then
$y \leftarrow FHE.Eval(hpk, SE.Enc,$
$FHE.Enc(sk), p[m\lambda/2 + 1 : m\lambda])$
Else $y \leftarrow FHE.Eval(hpk, SE.Enc,$
$FHE.Enc(sk), p[1 : m\lambda/2])$
EndIf
```

$(d, Cert) \leftarrow VS.Checker(i, p, y)$

```
If $d$ is the value the developer
tries to commit in the commit
stage, and $SE.Dec(sk, d) = Enc_x$,
then
If $Enc_x = \perp$ then
the verifier avoids evaluating any table that
has incoming edge from $PT_i'$ with $PT_i'(x^i)$ as input
Else if $Enc_x = \top$ then
the verifier can continue evaluating any table that
has incoming edge from $PT_i'$ with $PT_i'(x^i)$ as input
Else if $Enc_x$ is $null$ then
the verifier knows that the question
it asks the developer is illegal and
it should check the correctness of the question
End If
```

Figure 4.3: The protocol of VS in our implementation

4.2.3 Proof of the implementation correctness and security

In our implementation $CP' = CP'' = CP$. The input $Cerificate$ of $VS.Eval$ in Definition 20 in our implementation is (QA_E, QA_C, G', hpk, U).

In Definition 22, we have three simulators S_1, S_2 and S_3 simulating $VS.Encrypt$, $VS.Encode$ and $VS.Checker$. The input to simulator S_1 consists of the circuit size s and depth d of all the circuits resulting from the rhs functions in the tables in G as well as the structure graph G_{struc} of G. In our implementation we add another algorithm $VS.Path$, so we add another simulator S_4 to simulate $VS.Path$. We add an oracle O_3 for S_4, which is not in Definition 18.

Proof of the implementation security

First we prove that the verification scheme VS in Subsection 4.2.2 satisfies Definition 22. Besides, we need to point out that the developer in Exp^{real} and Exp^{ideal} of Definition 18 is an honest developer (see Remark 10).

Theorem 4. *The verification scheme VS introduced in Subsection 4.2.2 satisfies Definition 22.*

Proof. We construct a tuple of simulators (S_1, S_2, S_3, S_4) such that S_1,$S_2^{O_1}$ and $S_3^{O_2}$ are the same S_1,$S_2^{O_1}$ and $S_3^{O_2}$ in the proof of Theorem 2. S_4 receives queries from A_2 and queries oracle O_3, which is described in Algorithm 10. S_4 then returns the output of O_3 to A_2.

First we add an extra experiment $Exp^{extra}(1^K)$ as follows:

$Exp^{extra}(1^K)$:

1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
3. $a \leftarrow A_2^{VS.Encode, VS.Path,\ S_4^{O_3}}(VS.Eval, G', G, VGA, CP, hpk, state_A)$
4. Output a

$Exp^{real}(1^K)$ and $Exp^{extra}(1^K)$ only differ at step 3 where A_2 queries $VS.Checker$ in $Exp^{real}(1^K)$ and $S_4^{O_3}$ in $Exp^{extra}(1^K)$. If $VS.Checker$ and $S_4^{O_3}$ have the same input-output functionality, then $Exp^{real}(1^K)$ and $Exp^{extra}(1^K)$ cannot be distinguished computationally. Therefore we prove below that $VS.Checker$ and $S_4^{O_3}$ have the same input-output functionality.

For an input (i, p, y) to $VS.Checker$ and O_3, there are two cases.

Case 1: When $VS.Checker(i, p, y)$ outputs $null$, it is easy to see that $O_3(i, p, y)$ will also output $null$. To see this, note that the following situations in Algorithm 8 will cause $VS.Checker(i, p, y)$ to output $null$:

(1) The size of p or y is not correct. (see line 1).

(2) p is not generated by the evaluation of a table or $VS.Encode$. (see line 2).

(3) F_{secret} is wrong. (see line 13 and Section 4.2.2).

(4) y cannot be generated by $VS.Checker$ from knowing p and F_{secret} (see lines 15-28).

These four situations will also cause $O_3(i, p, y)$ in Algorithm 10 to output *null*: line 1 matches situation (1); line 3 matches situation (2); line 17 matches situation (3); lines 19-31 match situation (4).

Case 2: We examine the case where $VS.Checker(i, p, y)$ does not output *null*. Then the developer launches a bit commitment protocol with V. During the commit stage, the developer generates $Cert$ and in the end $VS.Checker$ outputs $d, Cert$ (see lines 8-10 and line 29 of Algorithm 8).

In the case where $O_3(i, p, y)$ does not output *null*, it first generates the same d as $VS.Checker(i, p, y)$ (see lines 6-11 of Algorithm 10). Then the bit commitment protocol is launched with the verifier. During the commit stage, $Cert$ is generated and in the end $O_3(i, p, y)$ outputs $d, Cert$ (see lines 12-16 and line 32).

Therefore $VS.Checker$ and $S_4^{O_3}$ have the same input-output functionality. Hence $Exp^{real}(1^K)$ and $Exp^{extra}(1^K)$ are computationally indistinguishable:

$$|Pr[D(Exp^{real}(1^K), 1^K) = 1] - Pr[D(Exp^{extra}(1^K), 1^K) = 1]| \leq negl(K) \tag{4.4}$$

Besides, we can construct two new experiments $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ as follows:

$Exp^{rtest}(1^K)$:

1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
3. $a \leftarrow A_2^{VS.Encode, VS.Path}(G', G, CP, hpk, U, state_A)$

```
4. Output $a$
```

```
$Exp^{itest}(1^K)$:
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G'', hpk, U) \leftarrow S_1(1^K, s, d, G_{struc})$
3. $a \leftarrow A_2^{S_2^{O_1}, S_3^{O_2}}(G'', G, CP, hpk, U, state_A)$
4. Output $a$
```

In fact, $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ are the experiments $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ in Definition 18 with $VS.Path$ added to $Exp^{real}(1^K)$ and $S_3^{O_2}$ added to $Exp^{ideal}(1^K)$) (see the first three paragraphs of Section 3.2.5). Thus from Theorem 2 we know that $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ are computationally indistinguishable. Namely, for all pairs of p.p.t. adversaries $A = (A_1, A_2)$ and p.p.t. distinguisher D,

$$|Pr[D(Exp^{rtest}(1^K), 1^K) = 1] - Pr[D(Exp^{itest}(1^K), 1^K) = 1]| \leq negl(K) \tag{4.5}$$

Now we prove that $Exp^{ideal}(1^K)$ and $Exp^{extra}(1^K)$ are computationally indistinguishable by contradiction. First we assume that $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally distinguishable. Then from (4.4) we know that $Exp^{extra}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally distinguishable. In other words we know that $Exp^{ideal}(1^K)$ and $Exp^{extra}(1^K)$ must be computationally distinguishable by a p.p.t.

algorithm $\tilde{D}$ and p.p.t. adversary $\tilde{A} = (\tilde{A}_1, \tilde{A}_2)$ such that

$$|Pr[\tilde{D}(Exp^{ideal}(1^K), 1^K) = 1] - Pr[\tilde{D}(Exp^{extra}(1^K), 1^K) = 1]| > negl(K) \tag{4.6}$$

Now we show that for any p.p.t. adversary $A = (A_1, A_2)$ who wants to distinguish $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ by using a p.p.t. algorithm D, A can use $\tilde{A}$ and $\tilde{D}$ to achieve its goal. What A does is as follows:

1. A_1 runs $\tilde{A}_1$: $(G, CP, state_{\tilde{A}}) \leftarrow \tilde{A}_1(1^K)$
2. A_1 outputs $\tilde{A}_1$'s output: $(G, CP, state_{\tilde{A}}) \leftarrow A_1(1^K)$
3. A_2 either gets (G', hpk, U) from $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$ and has oracle access to $VS.Encode$.

   Or it gets (G'', hpk, U) from $(G'', hpk, U) \leftarrow S_1(1^K, s, d, G_{struc})$ and has oracle access to $S_2^{O_1}, S_3^{O_2}$

   Which one A_2 gets depends on which one of the two experiments ($Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$) $A = (A_1, A_2)$ is doing.
4. A_2 constructs $S_4^{O_3}$ and $VS.Eval$ which are also in $Exp^{ideal}(1^K)$ and $Exp^{extra}(1^K)$.
5. A_2 runs $\tilde{A}_2$ and provides it with oracle access to $S_4^{O_3}$:

   (1) $a \leftarrow \tilde{A}_2^{VS.Encode, VS.Path, S_4^{O_3}}(VS.Eval, G', G, CP, hpk, U, state_{\tilde{A}})$

   (2) $a \leftarrow \tilde{A}_2^{S_2^{O_1}, S_3^{O_2}, S_4^{O_3}}(VS.Eval, G'', G, CP, hpk, U, state_{\tilde{A}})$

Which one of the two above is actually executed depends on which one of the two experiments ($Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$) $A = (A_1, A_2)$ is trying to distinguish.

If A is trying to distinguish $Exp^{rtest}(1^K)$, then A will do step 1, 2, 3(1), 4, 5(1). We can easily see that

$$Pr[D(Exp^{rtes}(1^K)) = 1] = Pr[\tilde{D}(Exp^{extra}(1^K)) = 1.$$

If A is trying to distinguish $Exp^{itest}(1^K)$, then A will do steps 1, 2, 3(2), 4, 5(2). We can easily see that

$$Pr[D(Exp^{itest}(1^K)) = 1] = Pr[\tilde{D}(Exp^{ideal}(1^K)) = 1.$$

With (4.6), we know that

$$|Pr[D(Exp^{rtest}(1^K)) = 1] - Pr[D(Exp^{itest}(1^K)) = 1]| > negl(K).$$

However, this contradicts (4.5). Hence we know that the assumption that $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally distinguishable is wrong. Therefore, $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally indistinguishable. Thus we prove Theorem 4. □

Proof of the implementation correctness

Now we are going to prove that the verification scheme VS in subsection 4.2.2 satisfies Definition 21.

Theorem 5. *The verification scheme VS introduced in subsection 4.2.2 satisfies Definition 21.*

Proof. We prove that if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

then (4.1)-(4.3) are correct.

Now we first prove (4.2) is correct. We know that Qc_i is formatted as (k, p, y) and Ac_i is formatted as $(d, Cert)$. Similarly, Qe_i is formatted as $(k, x^k, PT'_k(x^k), q_2)$. (For simplicity, we only discuss the situation where Qe_i is formatted as $(k, x^k, PT'_k(x^k), q_2)$ and $PT'_k(x^k)$ is an external output. The other situations, where Qe_i is formatted as $(k, x^k_j, null, q_1)$ or $PT'_k(x^k)$ is not an external output, are similar to the situation we discuss below.)

There are three cases for the situation we discuss below.

Case 1: Case 1 discusses the situation where $VS.Encode$ and $VS.Encode'$ have the same input-output functionality. In this case, if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

we can be sure that for any input x of $VS.Checker$,

$$Pr[VS.Checker(x) = VS.Checker'(x)] \geq 1 - negl(K)$$

The reason is as follows: According to the construction of $VS.Eval$, it reads the question and answer set QA_C to check for any $(Qc_i, Ac_i, Sc_i) \in QA_C$, whether the verifier V has successfully launched and completed the bit commitment protocol described in Section 2.6 with the developer that runs $VS.Checker'$. We can model the interaction between V and the developer by running $VS.Checker'$ as follows:

(1) V sends Qc_i to the developer ($Qc_i = (k, p, y)$ and is the input to $VS.Checker'$).

(2) $VS.Checker'$ generates d' and the developer commits d' to V and it launches a bit commitment protocol with V. Sc_i is all the information both entities send to each other during the protocol. Sc_i includes the random vector $\vec{R}$ and $FHE.Enc(sk)$ (the secret key of the private key encryption scheme SE in the form of an FHE encryption). Sc_i also includes a random seed $Cert$ and Enc_d generated by $VS.Checker'$.

(3) Finally V gets the value that the developer wants to commit in the reveal stage of the bit commitment protocol, which is d. (V gets the output Ac_i of $VS.Checker'$. $Ac_i = (d, Cert)$.)

We know from Definition 2.6 that if the developer tries to reveal value other than the value it wants to commit in the commit stage, the probability it is going to succeed is negligible:

$$Pr[d = d'] \geq 1 - negl(K).$$

Now we discuss another subcase of Case 1, where for $(Qc_i, Ac_i, Sc_i) \in QA_C$, the value d that $VS.Checker'$ outputs is different than the value d^* which $VS.Checker$ would output. The interaction between V and the developer that runs $VS.Checker$ is as follows:

(1) V sends Qc_i to the developer ($Qc_i = (k, p, y)$ and is the input to $VS.Checker$).

(2) $VS.Checker$ generates d^* and the developer decides to commit d^* to V. It launches a bit commitment protocol with V and generates Sc'_i. Sc'_i includes

the random vector $\vec{R}$, $SE.Enc$,$FHE.Enc(sk)$, $Cert$ as the same $\vec{R}$, $SE.Enc$, $FHE.Enc(sk)$, and $Cert$ in Sc_i. Enc_{d^*} is also included in Sc_i'.

(3) Finally V gets the value that the developer wants to commit in the reveal stage of the bit commitment protocol, which is d^*. (V gets the output Ac_i^* of $VS.Checker$, $Ac_i^* = (d^*, Cert)$.)

We know that according to $VS.Eval$'s construction (see line 24 in Algorithm 9), it will decrypt d with SE's corresponding decryption algorithm $SE.Dec$ which outputs a new value $SE.Dec(sk, d)$. Then $VS.Eval$ will compare $SE.Dec(sk, d)$ with Ae_i from $(Qe_i, Ae_i) \in QA_E$, and if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

then we know (see line 24 in Algorithm 9)

$$SE.Dec(sk, d) = Ae_i \tag{4.7}$$

Below we prove

$$SE.Dec(sk, d^*) = Ae_i \tag{4.8}$$

According to $VS.Checker$'s construction (see lines 5-8 in Algorithm 8),

$$d^* = FHE.Dec(hsk, y), \tag{4.9}$$

where y is the third parameter of Qc_i.

Since $VS.Encode$ and $VS.Encode'$ have the same input-output functionality (which is mentioned at the beginning of Case 1), given $Qe_i = (k, x^k, PT_k'(x^k), q_2)$ as input to both $VS.Encode$ and $VS.Encode'$, the output of $VS.Encode$ and $VS.Encode'$ are the same. Namely, $Ae_i^* = Ae_i$, where Ae_i^* is the output of $VS.Encode$ and Ae_i is the output of $VS.Encode'$.

According to the construction of $VS.Encode$ (see lines 28-33 in Algorithm 2) ,

$$Ae_i^* = FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda]). \tag{4.10}$$

Therefore,

$$Ae_i = FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda]). \tag{4.11}$$

On the other hand, the relation between $PT_k'(x^k)$ and y is as the following (see Figure 4.3):

$$\begin{aligned} y =& FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), PT_k'(x^k)[m\lambda/2+1 : m\lambda]) \\ =& FHE.Enc(hpk, SE.Enc(sk, FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda])) \end{aligned} \tag{4.12}$$

((4.12) is valid according to Definition 11.)

Hence, by combining (4.9) and (4.12)

$$d^* = SE.Enc(sk, FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda]))) \tag{4.13}$$

Moreover, because of (4.13)

$$\begin{aligned} SE.Dec(sk, d^*) =& SE.Dec(sk, SE.Enc(sk, FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda])))) \\ =& FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2+1 : m\lambda]) \end{aligned} \tag{4.14}$$

Then, by combining (4.11) and (4.14), (4.8) is proved.

By combining (4.7) and (4.8), we know $d = d^*$ according to Definition 8. Thus, we proved that given input Qc_i to $VS.Checker$ and $VS.Checker'$, the output $Ac_i^* = (d^*, Cert)$ of $VS.Checker'$ and the output $Ac_i' = (d, Cert)$ of $VS.Checker$ are the same.

Therefore, we know that when $VS.Encode$ and $VS.Encode'$ have the same input-output functionality, if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

then for any input x of $VS.Checker$,

$$Pr[VS.Checker(x) = VS.Checker'(x)] \geq 1 - negl(K).$$

(The negligible error $negl(K)$ is caused by the FHE scheme (see Definition 11) and the bit commitment protocol (see Definition 15).) Hence, when $VS.Encode$ and $VS.Encode'$ have the same input-output functionality, (4.2) is proved.

Case 2: Case 2 discusses the situation where $VS.Checker$ and $VS.Checker'$ have the same input-output functionality. In this case, if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

we can be sure that for any input x of $VS.Encode$,

$$Pr[VS.Encode(x) = VS.Encode'(x)] \geq 1 - negl(K)$$

The reason is as follows: For $(Qe_i, Ae_i) \in QA_E$, the corresponding pair in QA_C is (Qc_i, Ac_i). $Qe_i = (k, x^k, PT'_k(x^k), q_2)$ is the input of $VS.Encode'$ and Ae_i is the output of $VS.Encode'$. $Qc_i = (i, p, y)$ is the input of $VS.Checker'$ and $Ac_i = (d, Cert)$ is the output of $VS.Checker'$. The relation of Qc_i and Qe_i is shown by (4.12). Because

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

we also know (4.7) is true (see line 24 in Algorithm 9).

On the other hand, because Ae_i^* is the output of $VS.Encode$, given Qe_i as the input to $VS.Encode$, (4.10) is true.

Moreover, according to $VS.Checker$'s construction (see lines 5-8 in Algorithm 8), given Qc_i as input, (4.9) is true.

In this case, $VS.Checker$ and $VS.Checker'$ have the same input-output functionality. We also just proved that (4.9) is true. Then we have

$$d = FHE.Dec(hsk, y) \tag{4.15}$$

Hence, by combining (4.7),(4.15),(4.12) we have

$$\begin{aligned} Ae_i =& SE.Dec(sk, d) \\ =& SE.Dec(sk, FHE.Dec(hsk, y)) \\ =& SE.Dec(sk, SE.Enc(sk, FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2 + 1 : m\lambda]))) \\ =& FHE.Dec(hsk, PT_k'(x^k)[m\lambda/2 + 1 : m\lambda]) \end{aligned}$$

Namely, we prove (4.11) is true. Then, by combining (4.11) and (4.10), we prove $Ae_i = Ae_i^*$. Namely, we proved that given Qe_i as input to $VS.Encode$ and $VS.Encode'$, their outputs are the same.

Therefore, we know that when $VS.Checker$ and $VS.Checker'$ have the same input-output functionality, if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

then for any input x of $VS.Encode$,

$$Pr[VS.Encode(x) = VS.Encode'(x)] \geq 1 - negl(K).$$

(The negligible error $negl(K)$ is caused by the FHE scheme (see Definition 11).) Hence, when $VS.Checker$ and $VS.Checker'$ have the same input-output functionality, (4.2) is proved .

Case 3: Finally we consider the case where there exists at least an input Qc_i such that $VS.Checker(Qc_i) \neq VS.Checker'(Qc_i)$ and there exists at least an input Qe_j such that $VS.Encode(Qe_j) \neq VS.Encode'(Qe_j)'$. We are going to prove below that when

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

(4.2) is true. First suppose $i \neq j$.

Given (Qe_j, Ae_j) and (Qc_j, Ac_j, Sc_j), if $VS.Checker(Qc_j) = VS.Checker'(Qc_j)$, then this subcase falls into Case 2. From Case 2 we know that for (Qe_j, Ae_j) and (Qc_j, Ac_j, Sc_j), if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

$$Pr[VS.Encode(Qe_j) = VS.Encode'(Qe_j), \\ VS.Checker(Qc_j) = VS.Checker'(Qc_j)] \geq 1 - negl(K) \quad (4.16)$$

Similarly, given (Qc_i, Ac_i, Sc_j) and (Qe_i, Ae_i), if $VS.Encode(Qe_i) = VS.Encode'(Qe_i)$, then this subcase falls into Case 1. From Case 1 we know that for (Qe_i, Ae_i) and (Qc_i, Ac_i, Sc_i), if

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

$$Pr[VS.Encode(Qe_i) = VS.Encode'(Qe_i), \\ VS.Checker(Qc_i) = VS.Checker'(Qc_i)] \geq 1 - negl(K) \quad (4.17)$$

Hence, the subcase we discuss below is $i = j$. Namely, the case where there exist at least an input Qc_i and the corresponding Qe_i such that $VS.Checker(Qc_i) \neq VS.Checker'(Qc_i)$ and $VS.Encode(Qe_i) \neq VS.Encode(Qe_i)'$.

On the other hand, we know from the construction of $VS.Encode$ (see lines 28-33 in Algorithm 2) that (4.10) is true.

Moreover, (4.12) is also proved in Case 1.

Therefore, by combining (4.9), (4.10) and (4.12), we have

$$\begin{aligned} SE.Dec(sk, d^*) =& SE.Dec(sk, FHE.Dec(hsk, y)) \\ =& SE.Dec(sk, SE.Enc(sk, FHE.Dec(hsk, PT'_k(x^k)[m\lambda/2 + 1 : m\lambda]))) \\ =& FHE.Dec(hsk, PT'_k(x^k)[m\lambda/2 + 1 : m\lambda]) \\ =& Ae_i^* \end{aligned} \tag{4.18}$$

Because $d^* \neq d$ and $Ae_i^* \neq Ae_i$, d and Ae_i do not necessarily satisfy (4.7). According to the interaction between V and the developer that runs $VS.Checker'$ in Case 1, $VS.Checker'$ only knows the secret key sk of SE generated by V after $VS.Checker'$ generates d. Therefore, according to Definition 8, the probability of correctly creating a ciphertext of Ae_i (which is d) without knowing the secret key sk is negligible.

$$Pr[SE.Dec(sk, d) = Ae_i] = Pr[SE.Enc(sk, Ae_i) = d] \leq negl(K) \tag{4.19}$$

However, according to $VS.Eval$'s construction (see line 24 in Algorithm 9), when

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

(4.7) is true. Thus, (4.7) contradicts with (4.19). Therefore we know that there does not exist an input Qc_i and the corresponding Qe_i such that

$$\begin{aligned} &Pr[VS.Checker(Qc_i) \neq VS.Checker'(Qc_i), \\ &\qquad VS.Encode(Qe_i) \neq VS.Encode(Qe_i)'] \geq negl(K). \end{aligned}$$

In other words, for any Qc_i and the corresponding Qe_i we have

$$\begin{aligned} &Pr[VS.Checker(Qc_i) = VS.Checker'(Qc_i) \text{ or} \\ &\qquad VS.Encode(Qe_i) = VS.Encode(Qe_i)'] \geq 1 - negl(K). \end{aligned} \tag{4.20}$$

Therefore, by combining (4.16), (4.17) and (4.20), we know that when

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

(4.2) is correct.

Now we are going to prove that when

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

(4.3) is correct.

Because according to $VS.Eval$'s construction in Algorithm 9 (see line 1 in Algorithm 9), $VS.Eval$ of Algorithm 9 outputting 1 implies $VS.Eval$ of Algorithm 3 outputting 1. Thus, we know (3.1) is true. (In (3.1), V can only ask the developer to run $VS.Encode$ and $VS.Path$.) We use $V^{VS.Checker}$ to denote that V can ask the developer to run $VS.Checker$. If V can ask the developer to run $VS.Checker$, (3.1) still holds. (3.1) will become the following:

$$Pr_r[V(1^K, G, F_{spec}, VGA_r, CP)(r) = V^{VS.Checker}(1^K, G', F_{spec}, VGA_r, CP)(r)] \geq 1 - negl(K), \quad (4.21)$$

The reason that (4.21) holds is as follows: $VS.Checker$'s job is to check whether the developer runs $VS.Encode$ (see Case 2 of the proof of Theorem 5) and when $VS.Eval$ of Algorithm 9 outputting 1, V follows the protocol of Figure 4.3 and knows that the developer runs $VS.Encode$. The rest part of what V does in the protocol of Figure 4.3 is the same as what V does in the protocol of Figure 3.2. Therefore, $V^{VS.Checker}(1^K, G', F_{spec}, VGA_r, CP)(r)$ in (4.21) outputs the same as what $V(1^K, G', F_{spec}, VGA_r, CP)(r)$ outputs in (3.1).

We have already proved that when

$$VS.Eval(1^k, QA_E, QA_C, G', hpk, U) = 1,$$

(4.2) is correct. Thus with (4.21), we know that the following is true:

$$Pr_r[V(1^K, G, F_{spec}, VGA_r, CP)(r) = V^{VS.Checker'}(1^K, G', F_{spec}, VGA_r, CP)(r)] \geq 1 - negl(K), \quad (4.22)$$

where in the above equation the developer runs $VS.Checker'$ and $VS.Encode'$. Because (4.22) is (4.3), thus (4.3) is true.

Similarly, we can prove that (4.1) is true.

Thus, we proved Definition 21.

□

Theorem 6. *The verification scheme VS in subsection 4.2.1 and 4.2.2 satisfies Definition 20.*

Proof. Theorem 5 and Theorem 4 imply Theorem 6. □

Algorithm 8 $VS.Checker(i, p, y)$

```
1: if $|y| \neq m\lambda$ or $|p| \neq m\lambda$ then return null
2: else if ($\nexists(Q_k, A_k) \in QA_E$ such that $Q_k = (i, x^i, PT_i'(x^i), q_2)$ and $p = PT_i'(x^i)$ or $\nexists(Qe_k, Ae_k) \in QA_E$ such that $(Qe_k, Ae_k) = ((i, u_j^i, null, q_1), w_j^i)$ and $p = w_j^i$ then
3:     return null
4: else
5:     for all $i \in \{0, ..., m-1\}$ do
6:         $b_{i+1} \leftarrow FHE.Dec(hsk, y[i * \lambda + 1 : (i+1) * \lambda])$
7:     end for
8:     The developer starts the bit commitment protocol described in Section 2.6. The developer wants to commit to the verifier $d = b_1, ..., b_m$. The developer asks $V$ to provide $R$ as specified in commit stage (1)
9:     $V$ sends $R$ to the developer
10:    The developer does its part in commit stage (2) and sends to $V$ $Enc_d$
11:    The developer asks for $V$ to send $F_{secret}$
12:    $V$ sends $F_{secret}$ to the developer
13:    if $F_{secret} \neq (SE.Enc, FHE.Enc(hpk, sk))$ where $sk$ is a secret key of $SE$ then return null
14:    end if
15:    if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, x^i, PT_i'(x^i), q_2)$, $p = PT_i'(x^i)$ and $PT_i'$'s output is an intermediate output according to $G_{struc}$ then
16:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[1 : m/2 * \lambda]) \neq y$ then
17:            return $null$
18:        end if
19:    else if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, u_j^i, null, q_1)$, $Ae_k = p$ and $PT_i'$'s input is an external input according to $G_{struc}$ then
20:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p) \neq y$ then
21:            return $null$
22:        end if
23:    else
24:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[m/2 * \lambda + 1 : m * \lambda]) \neq y$ then
25:            return $null$
26:        end if
27:    end if
28: end if
29: return $d, Cert$
```

Algorithm 9 $VS.Eval(1^K, QA_E, QA_C, G', hpk, U)$

```
1: if $VS.Eval(1^K, QA_E, G', hpk, U)$=0 then /* $VS.Eval$ in this line refers to Algorithm 3 */
2:     return 0
3: end if
4: for all $(Qe_i, Ae_i) \in QA_E$ do
5:     if $\nexists(Qc_i, Ac_i, Sci) \in QA_C$ then return 0
6:     else find the corresponding $(Qc_i, Ac_i, Sc_i) \in QA_C$ where $Qc_i = (i, p, y)$, $Ac_i = (d', Cert), Sc_i = (SE.Enc, FHE.Enc(hpk, sk), Enc_d, R)$
7:     end if
8:     $V$ does Bob's part in the revealing stage in Definition 2.6, taking $Cert, Enc_d, R, d'$ as input and checks whether $d'$ is the value $d$ committed to $V$ in the commit stage
9:     if $VS.Checker$ tries to reveal a different sequence of bits $d'$ other than $d$ then return 0
10:    end if
11:    if $Qe_i$ is $(i, v_j^i, null, q_1)$ then
12:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p) \neq y$ then return 0
13:        end if
14:        if $SE.Dec(sk, d) \neq v_j^i$ then return 0
15:        else return 1
16:        end if
17:    else if $Qe_i$ is $(i, x^i, PT_i'(x^i), q_2)$ and $PT_i'(x^i)$ is an external output then
18:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[m\lambda/2 + 1 : m\lambda]) \neq y$ then return 0
19:        else if $SE.Dec(sk, d) \neq Ae_i$ then return 0
20:        else return 1
21:        end if
22:    else if $Qe_i$ is $(i, x^i, PT_i'(x^i), q_2)$ and $PT_i'(x^i)$ is an intermediate output then
23:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[1 : m\lambda/2]) \neq y$ then return 0
24:        else if $SE.Dec(sk, d) \neq Ae_i$ then return 0
25:        else return 1
26:        end if
27:    end if
28: end for
```

Algorithm 10 $O_3(i, p, y)$

```
1: if $|y| \neq m\lambda$ or $|p| \neq m\lambda$ then
2:     return $null$
3: else if $\nexists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, x^i, PT'_i(x^i), q_2)$ and $p = PT'_i(x^i)$ or $\nexists(Qe_k, Ae_k) \in QA_E$ such that $(Qe_k, Ae_k) = ((i, u^i_j, null, q_1), w^i_j)$ and $p = w^i_j$ then
4:     return $null$
5: else
6:     $O_3$ finds out the pair $(Qe_k, Ae_k)$ from $QA_E$ that matches the input $(i, p, y)$.
7:     if $Qe_k$ is $(i, u^i_j, null, q_1)$ then
8:         $a \leftarrow u^i_j$.
9:     else $a \leftarrow Ae_k$.
10:    end if
11:    Because $O_3$ knows the secret of the adversary A, it knows the secret key $sk$ and $SE.Enc$. Thus, $b_1b_2...b_m \leftarrow SE.Enc(sk, a)$
12:    The developer starts bit commitment protocol described in Section 2.6. The developer wants to commit to $V$ $d = b_1, ..., b_m$ and asks $V$ to provide $R$ as specified in commit stage (1)
13:    $V$ sends $R$ to the developer
14:    The developer does its part in commit stage (2) and generates $Cert$. It sends $Enc_d$ to $V$
15:    The developer asks for $V$ to send $F_{secret}$
16:    $V$ sends $F_{secret}$ to the developer
17:    if $F_{secret} \neq (SE.Enc, FHE.Enc(hpk, sk))$ then return $null$
18:    end if
19:    if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_t = (i, x^i, PT'_i(x^i), q_2)$, $p = PT'_i(x^i)$ and $PT'_i$'s output is an intermediate output according to $G_{struc}$ then
20:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[1 : m/2 * \lambda]) \neq y$ then
21:            return $null$
22:        end if
23:    else if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, u^i_j, null, q_1)$, $Ae_k = p$ and $PT'_i$'s input is an external input according to $G_{struc}$ then
24:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p) \neq y$ then
25:            return $null$
26:        end if
27:    else
28:        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[m/2 * \lambda + 1 : m * \lambda]) \neq y$ then
29:            return $null$
30:        end if
31:    end if
32:    return $d, Cert$
33: end if
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this work we studied the problem of protecting non-disclosed information of a program during the verification of the program. We proposed the concept of Secure and Trusted Verification (STV). The scheme provides protection of non-disclosed information about the program from the verifier and prevents malicious activities on both the developer and the verifier's side. Moreover, we came up with an implementation of STV and showed the security of this implementation. Nevertheless, there are still some open problems which we should review in the following section.

5.2 Future work

5.2.1 Improving efficiency

Yao's garbled circuits

In this thesis, we came up with an implementation of STV by using fully homomorphic encryption (FHE) to encrypt the tables. Applying our implementation to a large table graph may involve many times of FHE. Although FHE is a very powerful tool, its efficiency is a problem. Hence efficiency will be a big problem for our implementation in practical applications. Thus, in order to improve the efficiency of our implementation, there are two research directions: Either we come up with faster FHE schemes, or replace FHE with some other cryptographic primitives that are more efficient (and probably the security of the scheme is weakened). In this section, we mainly discuss the possibility of replacing FHE with other cryptographic primitives in our implementation.

As mentioned in Section 1.2, Yao's garbled circuit (Yao, 1982) is a way to hide a circuit, while allowing anyone to evaluate the garbled circuit. Given a circuit C with an input x, it generates a garbled circuit C' and an encoded input x'. Evaluation of $C'(x')$ will generate $C(x)$ without leaking any information about C or x other than $C(x)$. An intuition to apply Yao's garbled circuit to our implementation is for the developer to construct a circuit C for every rhs function f of the tables in the table graph G and then garble these circuits. In this way the developer constructs a new table graph G'. Then the developer gives G' with the garbled circuits to the verifier and let the verifier evaluate. Whenever the verifier chooses an input x to a table (whose rhs garbled circuit is C and the original rhs function is f), it asks the developer for an encoded input $Enc(x)$ of this input x. Then the verifier can evaluate the table with the encoded input $Enc(x)$ and get the output, which will be $C(x)$. In this way we hide the rhs functions of every table. But there are still two unsolved problems: First, the intermediate outputs of G' leak the intermediate outputs of G, which makes this implementation less secure. Second, evaluating the same garbled circuit with more than one encoded input would compromise the security of the garbled circuit, hence the verifier is only allowed to evaluate a garbled circuit with only one encoded input, which is a huge restriction.

Goldwasser et al. in (Goldwasser *et al.*, 2013) came up with a construction of *reusable garbled circuit* that allows many time evaluation of the garbled circuit. Though this does solve the one-time usage problem above, the construction itself includes FHE. Therefore applying this technique to our implementation will end up with the same efficiency problem.

On the other hand, the advantage of garbled circuits is that it is usually considered to be more efficient than FHE schemes. In (Huang *et al.*, 2011) the efficiency of garbled circuits and a homomorphic encryption scheme were compared in Hamming distance computation where garbled circuit technique won by a great margin. Therefore, using garbled circuits to hide information is a future problem worth studying. Though we may come up with a garbled circuits construction with less security, achieving better efficiency is worth the effort.

Verifiable computing

In Section 1.2 we mentioned verifiable computing. A series of work ((Cormode *et al.*, 2012), (Thaler *et al.*, 2012), (Vu *et al.*, 2013), (Setty *et al.*, 2012a), (Setty *et al.*, 2012b), (Parno *et al.*, 2013), (Ben-Sasson *et al.*, 2013)) implemented verifiable computing and achieved "near practical performance" (Walfish and Blumberg, 2015). In our current implementation in Chapter 3 and 4, in order to check the correctness of the verifier's evaluation of the program, repeating the whole evaluation is needed, which in some occasion is impractical. If we can somehow come up with a method to hide the computation in verifiable computing, then it can be applied to our implementation of secure and trusted verification and probably we can achieve better efficiency, because it generates verifiable computation results whose correctness can be verified without repeating the whole computation.

5.2.2 Hiding the graph structure

In our implementation of Trustworthy Verification of Non-disclosed Information in Section 3.2 and 4.2, the graph structure of the original table graph G is completely revealed in the encrypted table graph G'. As stated in Section 3.2, revealing the graph structure is necessary in order to do more than just black box verification. However, making that compromise does make our implementation less secure than encrypting the whole implementation into one table. The verifier may figure out the content of the table graph G by merely observing the graph structure of G. For example, suppose G uses a widely-used algorithm A whose table graph is a subgraph of G and the verifier knows this subgraph. Then if the verifier finds out that this subgraph of G highly resembles the table graph of A, it may figure out that G includes algorithm A, which however, should be kept secret. Thus, hiding the graph structure of G is not a trivial problem.

On the other hand, as already mentioned, the idea of condensing the table graph

G into one big table is not a good solution to hiding G's graph structure. The verifier needs G's graph structure to do non-black box verification.

We propose the idea of creating a larger table graph LG such that the table graph G becomes its subgraph or will become its subgraph after certain transformation. The encrypted version of LG, which we denote as LG', will reveal the graph structure of LG. And by using LG' to do the verification, the verifier does the verification of G', which is the encrypted version of G. Moreover, though LG includes information about G's graph structure, there may be ways to hide G inside LG such that it is very hard to figure out G's graph structure from LG's graph structure.

A similar problem exists in social network systems and databases. It is the problem of identity anonymization on graphs. For example, a database of social connections in a social network may need to be released for research. The connections can be modelled as a graph, and, before the release, the identities of the nodes of the social network graph need to be anonymized. However, merely deleting information about the nodes themselves may not be secure enough, because the graph structure may reveal information about the identity of the node.

There are already some papers on how to hide the graph structure in order to anonymize the identity of the nodes, e.g. (Zhou and Pei, 2008) and (Cheng *et al.*, 2010). We believe methods proposed in these papers may be useful to us.

Bibliography

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 cryptologyCRYPTO 2001*, pages 1–18. Springer.

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.

Cheng, J., Fu, A. W.-c., and Liu, J. (2010). K-isomorphism: privacy preserving network publication against structural attacks. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 459–470. ACM.

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand.

Cormode, G., Mitzenmacher, M., and Thaler, J. (2012). Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112. ACM.

Gentry, C. (2009). *A fully homomorphic encryption scheme*. Ph.D. thesis, Stanford University.

Gentry, C., Halevi, S., and Vaikuntanathan, V. (2010). i-hop homomorphic encryption and rerandomizable yao circuits. In *Advances in Cryptology–CRYPTO 2010*, pages 155–172. Springer.

Goldwasser, S., Kalai, Y. T., and Rothblum, G. N. (2008). One-time programs. In *Advances in Cryptology–CRYPTO 2008*, pages 39–56. Springer.

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.

Katz, J. and Lindell, Y. (2014). *Introduction to modern cryptography*. CRC Press.

Kilian, J. (1989). *Uses of randomness in algorithms and protocols*. Ph.D. thesis, Massachusetts Institute of Technology.

Lynn, B., Prabhakaran, M., and Sahai, A. (2004). Positive results and techniques for obfuscation. In *Advances in Cryptology-EUROCRYPT 2004*, pages 20–39. Springer.

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.

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

Parno, B., Howell, J., Gentry, C., and Raykova, M. (2013). Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE.

Sander, T., Young, A., and Yung, M. (1999). Non-interactive cryptocomputing for nc 1. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 554–566. IEEE.

Setty, S. T., McPherson, R., Blumberg, A. J., and Walfish, M. (2012a). Making argument systems for outsourced computation practical (sometimes). In *NDSS*.

Setty, S. T., Vu, V., Panpalia, N., Braun, B., Blumberg, A. J., and Walfish, M. (2012b). Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*, pages 253–268.

Standard, D. E. (1977). Fips 46. *NBS (Jan. 77)*.

Thaler, J., Roberts, M., Mitzenmacher, M., and Pfister, H. (2012). Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*.

Vaikuntanathan, V. (2011). Computing blindfolded: New developments in fully homomorphic encryption. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 5–16. IEEE.

Valiant, L. G. (1976). Universal circuits (preliminary report). In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 196–203. ACM.

Vu, V., Setty, S., Blumberg, A. J., and Walfish, M. (2013). A hybrid architecture for interactive verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 223–237. IEEE.

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

Wang, C., Hill, J., Knight, J., and Davidson, J. (2000). Software tamper resistance: Obstructing static analysis of programs. Technical report, Technical Report CS-2000-12, University of Virginia, 12 2000.

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

Yao, A. C. (1982). Protocols for secure computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE.

Zhou, B. and Pei, J. (2008). Preserving privacy in social networks against neighborhood attacks. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 506–515. IEEE.