Noname manuscript No.
(will be inserted by the editor)

Secure and Trusted Partial Grey-Box Verification

Yixian Cai ·
George Karakostas ·
Alan Wassyng

the date of receipt and acceptance should be inserted later

Abstract A crucial aspect in the development of software-intensive systems is *verification*. This is the process of checking whether the system has been implemented in compliance with its specification. In many situations, the manufacture of one or more components of the system is outsourced. We study the case of how a third party (the *verifier*) can verify an outsourced component effectively, without access to all the details of the internal design of that component built by the *developer*. We limit the design detail that is made available to the verifier to a diagram of interconnections between the different design units within the component, but encrypt the design details within the units and also the intermediate values passed between the design units. We formalize this notion of limited information using tabular expressions to describe the functions in both the specifications and the design. The most common form of verification is testing, and it is known that black-box testing of the component is not effective enough in deriving test cases that will adequately determine the correctness of the implementation, and the safety of its behaviour. We have developed protocols that allow for the derivation of test cases that take advantage of the design details disclosed as described above. We can regard this as *partial grey-box testing* that does not compromise the developer's secret information. Our protocols work with both trusted and untrusted developers, as well as trusted and untrusted verifiers, and allow for the checking of the correctness of the verification process itself by any third party, and at any time. Currently our results are derived under the simplifying assumption that the software design units are linked acyclically. We leave the lifting of this assumption as an open problem for future research.

1 Introduction

This paper deals with protection of the intellectual property embedded in a software component. A typical scenario is that the manufacturer/integrator of the system provides a requirements specification of a component to a sub-contractor, and the sub-contractor eventually delivers the component to the system manufacturer. It is common practice for the manufacturer to then conduct acceptance testing, that is typically black-box testing, since the sub-contractor may not want to disclose information that could compromise their intellectual property. However, black-box testing is likely not to be satisfactory in discovering safety, security and dependability flaws in the system, introduced by that component. What the manufacturer would be better-off doing, is to perform grey-box testing. In other words, derive test cases through knowledge of both the requirements specification, and the actual software design/implementation of the component.

The following scenario is a rather common example of the design and development of a product: A car manufacturer needs to incorporate a fuel delivery component, made by a sub-contracting party, into its car design. In order to do that, it first produces the *requirements specifications* of the component (described in a standard way), and provides them to the sub-contractor. The latter implements its own design, and delivers the component, claiming that it complies with the specifications. The car manufacturer, in turn, *verifies* that the claim is true, by running a set of *test cases* on the component, and confirming that the outputs produced for these test inputs are consistent with the specifications. After the car design has been completed, a government agency may need to approve it, by running its own tests and confirming that the outputs comply with its own publicly-known set of environmental specifications; once this happens, the car can be mass-produced and is allowed on the road. The only problem is that both the car manufacturer and the sub-contractor *do not want to reveal* their designs to any third party, since they contain proprietary information, such as the setting of certain parameters that took years of experimentation to fine tune; therefore, with only the requirements specifications *publicly* available, only black-box testing can be performed by the car manufacturer (vis-a-vis the sub-contractor's component), and the government agency (vis-a-vis the car). It may be the case, though, that the car manufacturer and/or the sub-contractor are willing to reveal some partial information about their implementation(s) (possibly information that industry experts would figure out anyways given some time). In this case, an obvious question is the following: "Can we allow for testing, that possibly takes advantage of this extra information, *without revealing any information beyond*

Yixian Cai, George Karakostas, Alan Wassyng
McMaster University,
Dept. of Computing and Software,
1280 Main St. West, Hamilton, Ontario L8S 4K1, Canada,
E-mail: {caiy35,karakos,wassyng}@mcmaster.ca

that?” The answer to this question is rather obvious for the two extremes of (i) completely hidden information (only *black-box* testing is possible), and (ii) completely revealed information (*complete grey-box* testing is possible). In this work, we present initial steps in developing a theory and implementation that can answer the question in the affirmative – in the continuum between these two extremes, i.e., for partial grey-box testing.

Although our methods apply to any design whose specifications and component interaction can be described using a standard description method, like tabular expressions [1], for reasons of clarity we are going to restrict our discussion to the case of Software Engineering, i.e., to software products. There are two parties, i.e., a *software developer* (or *product designer* - we will use these terms interchangeably in what follows), and a *verifier*, as well as a publicly known set of (requirements) *specifications*, also in a standard description. The two parties engage in a protocol, whose end-result must be the *acceptance* or *rejection* of the implemented component by the verifier, and satisfies the following two properties:

- **Correctness:** If the two parties follow the protocol, and all test cases produced and tried by the verifier comply with the specifications, then the verifier accepts, otherwise it rejects.
- **Security:** By following the protocol, the developer does not reveal any information over-and-above that information the developer had agreed to disclose.

In this setting, the specifications may come from the developer, the verifier, or a third party, but are always public knowledge. (Public knowledge will mean that all parties – manufacturer, developer and regulator – are privy to it.) The algorithm that produces the test cases tried by the verifier is run by the latter, and is also publicly known and chosen ahead of time – Modified Condition/Decision Coverage (MC/DC), for example. Depending on the application, the developer may be either *trusted* (*honest*), i.e., follows the protocol exactly, providing always the correct (encrypted) replies to the verifier’s queries, or *untrusted* (*dishonest*) otherwise, and similarly for the verifier.

In this work, first we give a concrete notion of *partial* grey-box testing, using tabular expressions as the description tool of choice. In its simpler form, a tabular expression is essentially a tabular notation that describes the output(s) of a function, given mutually exclusive predicates on the inputs of the function. Each row contains a left-hand side (LHS) predicate on the table inputs, and the right-hand side (RHS) contains the value of the output(s). The RHS of a row is the output value of the function if-and-only-if the LHS predicate is evaluated to TRUE. A table describes the functionality of a design component; the description of the whole design is done by constructing a *directed acyclic table graph*, representing the interconnection of tables (nodes of the graph) as the output of one is used as an input to another (edges of the graph). Our crucial assumption is that the table graph is exactly the *extra publicly-known information* made available by the developer (in addition to the single publicly-known table describing the specifications); the contents of the tables-nodes, as well as the actual intermediate table input/output values should remain *secret*. Our goal is to facilitate the testing and compliance verification of designs that are comprised of many components, which may be off-the-shelf or implemented by third parties; if these components are trusted to comply with their specifications by the developer (by, e.g., passing a similar verification process), then all the developer needs to know, in order to proceed with the verification of the whole design, is their input/output functionality, and not the particulars of their implementation. The formal definition of table graphs can be found in Section 2.1.

We present protocols that allow the verifier to run test case-generating algorithms that may take advantage of this extra available information (e.g., MC/DC [2]), and satisfy the correctness and security properties (formally defined in Section 2.2) with high probability (whp). We break the task of verification in a set of algorithms for the encryption of table contents and intermediate inputs/outputs (run by the developer), and an algorithm for checking the validity of the verification (cf. Definition 3); the former ensure the security property, while the latter will force the verifier to be honest, and allow any third party to verify the correctness of the verification process at any future time. Initially we present a protocol for the case of a trusted developer (Section 4), followed by a protocol for the case of a dishonest one (Section 5). Our main cryptographic tool is *Fully Homomorphic Encryption* (FHE), a powerful encryption concept that allows computation over encrypted data first implemented in [3]. We also employ bit-commitment protocols [4] in Section 5.

1.1 Previous work

The goal of *obfuscation*, i.e., hiding the code of a program, while maintaining its original functionality, is very natural, and has been the focus of research for a long time. An obfuscated program reveals no information about the original program, other than what can be figured out by having black-box access to the original program. There are many heuristic obfuscation methods, e.g., [5], [6]. However, as shown in [7], 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.

A crucial aspect of our work is the exclusion of any third-parties that act as authenticators, or guarantors, or a dedicated server controlled by both parties like the one in [8], called an ‘amanat’. In other words, we require that there is no shared resource between the two interacting parties, except the communication channel and the public specifications. This reduces the opportunities for attacks (e.g., a malicious agent taking over a server, or a malicious guarantor), since they are reduced to attacking the channel, a very-well and studied problem.

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* [9], which allows for the obfuscation of a point function (i.e., a function that outputs 1 for one specific input, and 0 otherwise), but not arbitrary functions. Another relevant cryptographic primitive is Yao’s *garbled circuits* [10], which allow the computation of a circuit’s output on a given input, without revealing any information about the circuit or the input. Yao’s garbled circuits nearly achieve what obfuscation requires in terms of one-time usage, and are the foundation of work like one-time programs [11]. Its problem is that the circuit encryption (i.e., the garbled circuit) can be run only once without compromising security. Recently, [12] proposed a new version of garbled circuits called *reusable garbled circuits*, which allows for a many-time usage on the same garbled circuit, and still satisfies the security of Yao’s garbled circuits. [12] then uses reusable garbled circuits to implement *token-based obfuscation*, i.e., obfuscation that allows a program user to run the obfuscated program on any input, without having to ask the developer who obfuscated the program to encode the input before it is used. Token-based obfuscation guarantees that only black-box information about the original program can be figured out from the obfuscated program. Unfortunately, when a program is token-based obfuscated, it becomes a black box to the user (due to the inherent nature of obfuscation), and thus precludes its use for any kind of grey-box verification. Building on [12], our work proposes a method to alleviate this weakness.

Plenty of work has been done in the field of *verifiable computing*, which is also relevant to our work. Verifiable computing allows one party (the prover) to generate the verifiable computation of a function, whose correctness, a second party (a verifier - not related to the verifier in our setting) can then verify. For example, a naive way to do this, is to ask the verifier to repeat the computation of the function; however, in many cases this solution is both inefficient and incorrect. Recently, works like [13], [14], [15], [16], [17], [18], [19] which are based on the PCP theorem [20], presented systems that allow the verifier to verify the computation result without re-executing the function. The difference between verifiable computing and our work is that the developer wishes to hide the implementation of a design, while for verifiable computing, the design implementation cannot be hidden. However, verifiable computing still has the potential to be applied to the implementation of secure and trusted verification.

2 Formal framework for grey-box verification

In this section we develop a formal framework for grey-box verification. We start by describing tabular expressions (*tables*) in Section 2.1, a classical format used for describing component requirements (specs), followed by the definition of a *table graph*, i.e., a graph describing the interconnections between the tabular expressions of different components. Then, in Section 2.2, we first define the *evaluation* of an input on a given table graph as checking whether the output corresponding to the input according to the specs is consistent with the implementation path(s) in the graph and its intermediate results. The evaluation of test inputs produced by a *testing algorithm* is the basic task performed by a trusted *verifier* (Definition 2), following a protocol (*verification scheme*), formally defined in Definition 3, that allows the encryption of the data provided by the developer, and their correct evaluation by the verifier. That ensures that the verification scheme is *secure and trusted* (Definition 6), i.e., it is both *correct* in its evaluation of an input (Definition 4), and *secure* by not leaking implementation information during the evaluation process (Definition 5).

2.1 Tabular Expressions (Tables)

Tabular expressions (or simply *tables*, as we are going to call them in this work) are used for the documentation of software programs (or, more generally, engineering designs). The concept was first introduced by David Parnas in the 1970s (cf. [1] and the references therein), and since then there has been a proliferation of different semantics and syntax variations developed. In this work we use a simple variation [21], already used in the development of critical software. Figure 1(a) describes the structure of such a table. The *Conditions* column contains predicates $p_i(x)$, and the *Functions* column contains functions $f_i(x)$ (constant values are regarded as zero-arity functions). The table works as follows: If $p_i(x) = True$ for an input x, then $f_i(x)$ will be the output $T(x)$ of table T. To work properly, the predicates in T must satisfy the *disjointness* and *completeness* properties:

- **Completeness:** $p_1(x) \vee p_2(x) \vee ... \vee p_n(x) = True$
- **Disjointness:** $p_i(x) \wedge p_j(x) = False, \forall i, j \in [n]$,

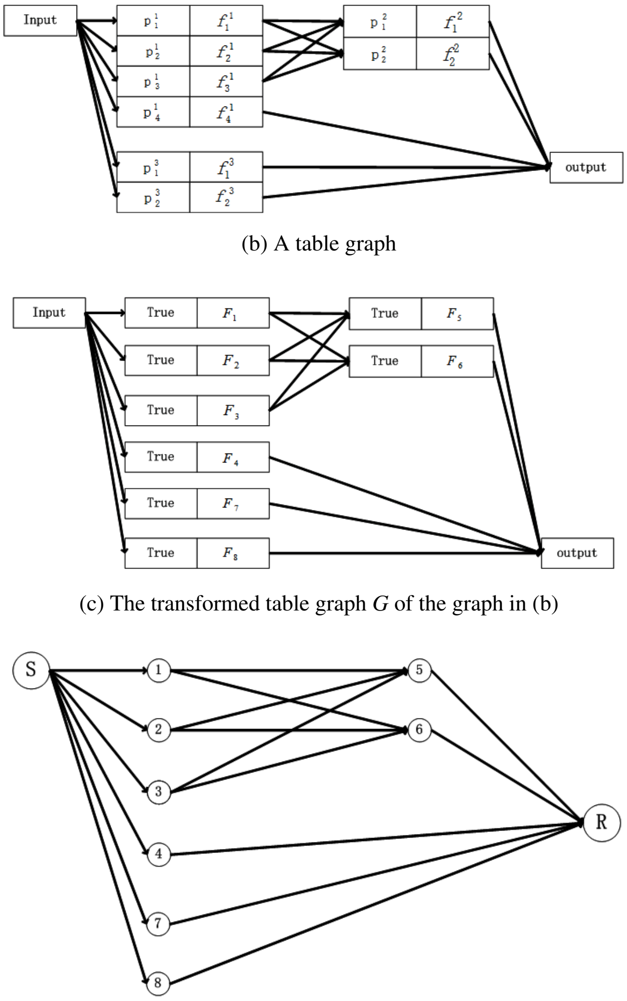
i.e., for any input x exactly one of the table predicates is $True$. Note that a table with a single row $(p_1(x), f_1(x))$ satisfies these properties only when $p_1(x) = True$, $\forall x$. This is important, because we are going to work with an equivalent representation that has only *single-row* tables.

$x \rightarrow$

Condition	Function
$p_1(x)$	$f_1(x)$
$p_2(x)$	$f_2(x)$
$\dots$	$\dots$
$p_n(x)$	$f_n(x)$

$\rightarrow T(x)$

(a) A table T with input x and output $T(x)$



(b) A table graph

(c) The transformed table graph G of the graph in (b)

(d) The structure graph G_{struc} of table graph G in (c)

Fig. 1: Tables, table graphs, and their transformation.

A table can be used to represent a module of a program (or a design). Then, the whole program can be documented as a directed *table graph*, with its different tables being the nodes of the graph, a source node *Input* representing the inputs to the program, and a sink node *Output* representing the outputs of the program. Every table is connected to the tables or *Input* where it gets its input from, and connecting to the tables or *Output*, depending on where its output is forwarded for further use. An example of such a table graph can be seen in Figure 1(b). We will make the following common assumption (achieved, for example, by loop urolling) for table graphs:

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

Note: This assumption should not be confused with a general assumption about loops within a design unit. The functionality achieved by a design unit is described by tabular expressions, which represent the behaviour of the function by the equivalent of pre- and post-conditions. The code implementation of such a tabular expression may contain loops.

In what follows, by *external input* we mean an input that comes directly from *Input*, and by *external output* we mean an output that goes directly to *Output*. An *intermediate input* will be an input to a table that is the output of another table, and an *intermediate output* will be an output of a table that is used as an input to another table.

Since Assumption 1 holds, we can inductively order the tables of a table graph G in *levels* as follows:

- An external input from node *Input* can be regarded as the output of a virtual level 0 table.
- A table is *at level* k ($k > 0$) if it has at least one incoming edge from a level $k-1$ table, and no incoming edges from tables of level larger than $k-1$.

As a consequence, we can traverse a table graph (or subgraph) in an order of increasing levels; we call such a traversal *consistent*, because it allows for the correct evaluation of the graph tables (when evaluation is possible), starting from level 0. The graph structure of a table graph G can be abstracted by a corresponding *structure graph* G_{struc}, which replaces each table in G with a simple node. For example, Figure 1(d) shows the structure graph for the table graph in Figure 1(c).

If $TS = \{T_1, T_2, \dots, T_n\}$ is the set of tables in the representation of a program, we will use the shorthand

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

to denote that its transformed table graph G contains both the table information of TS and the graph structure G_{struc}.

2.2 Evaluation and verification

In formally defining an evaluation and verification scheme for grey-box verification, and its properties, the following standard definition of negligible functions will be used:

Definition 1 *A function $f(x)$ is called* negligible *(and denoted as $negl(x)$) iff for all $c > 0$, there exists x_0 such that $f(x) < x^{-c}$, $\forall x > x_0$.*

Evaluation: Tabular expressions (tables) can be used to represent a program at different levels of abstraction. In this work, we concentrate on two levels: the *specifications* level, and the *implementation* level. Our goal is to design protocols that will allow a third party (usually called the *verifier* in our work) to securely and truthfully verify the compliance of a design at the implementation level (represented by a table graph G) with the publicly known specifications (represented by a table graph G^{spec}), without publicly disclosing more than the structure graph G_{struc} of G.

We are going to abuse notation, and use G^{spec} and G also as functions $G^{spec} : D^{spec} \to R^{spec}$, and $G : D \to R$, where D^{spec}, D are the external input domains, and R^{spec}, R are the external output ranges of table graphs G^{spec} and G, respectively. Without loss of generality, we assume that these two functions are over the same domain and have the same range, i.e., $D^{spec} = D$ and $R^{spec} = R$.[1]

The evaluation of G^{spec}, G (or any table graph), can be broken down into the evaluation of the individual tables level-by-level. The *evaluation* of a table T_i at input $x^i = (x^i_1, x^i_2, \ldots, x^i_k)$ produces an output $y^i = T_i(x^i)$. Note that some inputs x^i_j may be *null*, if they come from non-evaluated tables, and if T_i is not evaluated at all, then $T_i(x^i) = null$.

Given an external input X to the table graph G, the *evaluation* of $G(X)$, is done as follows: Let $TSO = (T_{s_1}, ..., T_{s_n})$ be an ordering of the tables in G in increasing level numbers (note that this is also a topological ordering, and that the first table is the virtual table of level 0). Let $T_{i_1}, ..., T_{i_s}$ be the tables that have external outputs. Then the evaluation of G is the evaluation of its tables in the TSO order, and $G(X)$ is the set of the external outputs of $T_{i_1}, ..., T_{i_s}$. We call this process of evaluating $G(X)$ *consistent*, because we will never try to evaluate a table with some input from a table that has not been already evaluated. All graph evaluations will be assumed to be consistent (otherwise they cannot be deterministically defined and then verified).

Verification: The verification processes we deal with here work in two phases:

- During the first phase, the table graph G is analyzed, and a set of (external or internal) inputs is generated, using a publicly known verification test-case generation algorithm VGA.
- In the second phase, G and G^{spec} are evaluated on the set of inputs generated during the first phase. If the evaluations coincide, we say that G *passes the verification*.

In our setting, VGA is any algorithm that takes the publicly known G_{struc}, G^{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). We emphasize that the output of VGA is not necessarily external inputs. Our protocols can also work with any VGA which generates enough information for the verifier to produce a corresponding set of external inputs, possibly by interacting with the designer of implementation G. We also allow for verification on a (possibly empty) predetermined and publicly known set CP of $(input, output)$ pairs, i.e., for every $(X,Y) \in CP$ with X being a subset of the external inputs and Y a subset of external outputs, G passes the verification iff $G(X)$ can be evaluated and $G(X) = Y$. The pairs in CP are called *critical points*.

Our protocols will use a *security parameter* K, represented in unary as K 1's, i.e., as 1^K. Now, we are ready to define a *trusted verifier* (or *trusted verification algorithm*) (we use the two terms interchangeably):

Definition 2 (Trusted verifier) *A* trusted verifier V *is a p.p.t. algorithm that uses r random bits, and such that*

$$Pr_r[V(1^K, G, G^{spec}, CP, VGA)(r) = \left. = \begin{cases} 0, \textit{ if } \exists X \in EI\text{: } G(X) \neq G^{spec}(X) \\ 0, \textit{ if } (CP \neq \emptyset) \wedge (\exists (X,Y) \in CP\text{: } G(X) \neq Y) \\ 1, \textit{ if the previous two conditions aren't true} \end{cases} \right] \geq 1 - negl(K)$$

where $EI \subseteq D$ is a (possibly empty) set of external inputs generated by V itself.

A verifier that satisfies Definition 2 is called trusted, because it behaves in the way a verifier is supposed to behave whp: Essentially, a trusted verifier uses the publicly known K, G^{spec}, CP, VGA and a publicly known G, to produce a set of external inputs EI; it accepts iff the verification does not fail at a point of EI or CP. During its running, the verifier can interact with the designer of G. While G is publicly known, the operations of the verifier are straight-forward, even without any interaction with the designer. The problem in our case is that the verifier has only an *encryption* of G, and yet, it needs to evaluate $G(X)$ in order to perform its test. This paper shows how to succeed in doing that, interacting with the designer of G, and without leaking more information about G than the publicly known G_{struc}.

In what follows, a verifier will be a part of a *verification scheme*, generally defined as follows:

Definition 3 (Verification scheme) *A* verification scheme *VS is a tuple of p.p.t. algorithms $(VS.Encrypt, VS.Encode, VS.Eval)$ such that*

- *$VS.Encrypt(1^K, G)$ is a p.p.t. algorithm that takes a security parameter 1^K and a table graph G as input, and outputs an encrypted table graph G'.*
- *$VS.Encode$ is a p.p.t. algorithm that takes an input x and returns an encoding Enc_x.*
- *$VS.Eval$ is a p.p.t. algorithm that takes a security parameter K and a public $Certificate$ as input, has an honest verifier V satisfying Definition 2 hardcoded in it, and outputs* 1 *if the verification has been done correctly (and* 0 *otherwise).*

[1] In general, the two pairs (D,R) and (D^{spec}, R^{spec}) may not be the same. Nevertheless, in this case there must be a predetermined mapping provided by the implementation designer, which converts (D_{spec}, R_{spec}) to (D,R) and vice versa, since, otherwise, the specifications verification is meaningless as a process.

From now on, it will be assumed (by the verifier and the general public) that the encryption by the designer is done (or *looks like* it has been done) by using a publicly known algorithm $VS.Encrypt$ (with a secret key, of course). If the designer's encryptions do not comply with the format of $VS.Encrypt$'s output, the verification fails automatically.

In general, it may be the case that an algorithm claiming to be a trusted verifier within a verification scheme, does not satisfy Definition 2 (either maliciously or unintentionally). Such a malicious verifier may claim that a design passes or does not pass the verification process, when the opposite is true. In order to guard against such a behaviour, we will require that there is a piece of public information, that will act as a *certificate*, and will allow the *exact* replication of the verification process by any other verifier and at any (possibly later) time; if this other verifier is a known *trusted* verifier, it will detect an incorrect verification with high probability (whp). We emphasize that the interaction (i.e., the messages exchange) shown in Figure 2 is done *publicly*. In fact, we will use the transaction record as a *certificate* (cf. Definition 3), that can be used to replicate and check the verification, as described above.

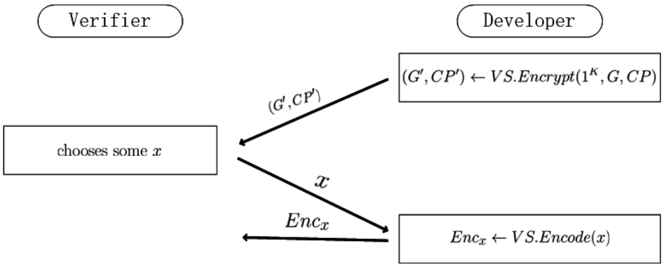

Verifier
Developer
$(G',CP') \leftarrow VS.Encrypt(1^K,G,CP)$
(G',CP')
chooses some x
x
Enc_x
$Enc_x \leftarrow VS.Encode(x)$

Fig. 2: The initial steps of the generic VS protocol in Definition 3.

Hence, we will require that our verification schemes are:

- *secure*, i.e., they do not leak the designer's private information, and
- *correct*, i.e., they produce the correct verification result whp.

More formally, we define:

Definition 4 (Correctness) *A verification scheme is* correct *iff the following holds: $VS.Eval(1^K, Certificate) = 1$ if and only if both of the following hold:*

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

$$Pr_r[V(1^K,G',G^{spec},VGA_r,CP)(r) = V'(1^K,G',G^{spec},VGA_r,CP)(r)] \geq 1 - negl(K) \tag{2}$$

where V is the honest verifier hardwired in $VS.Eval$, and G' is the table graph produced by $VS.Encrypt$.

Condition (2) forces V to be trusted, and condition (1) ensures that its verification is correct whp, even when applied to G' instead of G; together ensure the correctness of the verification whp.

Just as it is done in [12], we give a standard definition of security:

Definition 5 (Security) *For any two pairs of p.p.t. algorithms $A = (A_1, A_2)$ and $S = (S_1, S_2)$, consider the two experiments in Figure 3.*

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

(a) Experiment Exp^{real}

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

(b) Experiment Exp^{ideal}

Fig. 3: Experiments Exp^{real} and Exp^{ideal}

A verification scheme VS is secure *iff there exist a pair of p.p.t. algorithms (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 for any p.p.t. algorithm D:*

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

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

Definition 6 (Secure and trusted verification scheme) *A verification scheme of Definition 3 is* secure and trusted *iff it satisfies Definitions 4 and 5.*

Remark 1 *$VS.Encode$ and S_2^O in Step 3 of the two experiments in Figure 3 are* not *oracles used by A_2. In these experiments, A_2 plays the role of a (potentially malicious) verifier and interacts with the developer as shown in Figure 2. More specifically, in Exp^{real}, A_2 picks an input x,* asks the developer to run $VS.Encode(x)$ *(instead of querying an oracle), and receives the answer. In Exp^{ideal}, A_2 again asks the developer, but, unlike Exp^{real}, the latter runs S_2^O instead of $VS.Encode$ and provides A_2 with the answer. Hence, whenever we say that A_2 queries $VS.Encode$ or S_2^O, we mean that A_2 asks the developer to run $VS.Encode$ or S_2^O respectively,* *and provide the answer. Note that O* is *an oracle queried by* S_2.

In what follows, first we deal (in Section 4) with an *honest* designer/developer, i.e., a developer that answers honestly the verifier's queries:

Definition 7 *An* honest *developer is a developer that calls $VS.Encode(x)$ to generate Enc_x in Figure 2, when queried by the verifier.*

In the case of using the verifier to test a design, the developer obviously has no reason to not being honest. But in the case of verifying the correctness of the design, the developer may not follow Definition 7, in order to pass the verification process (cf. Section 5).

3 Preliminaries

In Section 3.1 we give a simple transformation of tables (and their table graphs) to single-row tables, and in Section 3.2 a standard encoding of table functions by boolean circuits, that will be then used by our implementation of a verification scheme. Then Section 3.3 enumerates some well-known cryptographic definitions and primitives, that will also be used by our implementation.

3.1 Transformation to single-row tables

Our algorithms will work with single row tables. Therefore, we show how to transform the general tables and graph in Figure 1(b) into an equivalent representation with one-row tables. The transformation is done in steps:

- As a first step, a table with n rows is broken down into n single-row tables. Note that, in general, the row of each multi-row table may receive different inputs and produce different outputs (with different destinations) that the other table rows; we keep the inputs and outputs of this row unchanged, when it becomes a separate table.
- We enhance each x_j of the external input $(x_1,x_2,...,x_s)$ (introduced to the program at node *Inputs*, and transmitted through outgoing edges to the rest of the original table graph) with a special symbol $\top$, to produce a new input $(x'_1,...,x'_s)$, with $x'_j = (\top,x_j)$, $j = 1,\ldots,s$.
- Let $(P(w),F(w))$ be the *(predicate, function)* pair for the new (single-row) table, corresponding to an old row $(p(x),f(x))$ of an original table. As noticed above, $P(w) = True$, no matter what the input $w = (w_1,w_2,\ldots,w_k)$ is. Let $x = (x_1,x_2,...,x_k)$, where $w_j = (\top,x_j)$ (i.e., the x_j's are the w_j's without the initial special symbol extension). The new function $F(w)$ is defined as follows:

$$F(w) = \begin{cases} (\top,f(x)), & \text{if } p(x) = 1 \\ (\bot,\bot), & \text{if } p(x) = 0. \end{cases}$$

Note that $\top$ and $\bot$ are special symbols, not related to the boolean values $True$ and $False$. In particular, from now on, we will assume that in case the first part of a table output is $\bot$, then this output will be recognized as a bogus argument, and will not be used as an input by a subsequent table. The reason for adding $\top,\bot$ to F 's output will become apparent in Section 4. If some x^i_j is $(\bot,\bot)$ or *null*, then we set $T_i(x^i) = null$.

After this transformation, we will get a transformed graph (Figure 1(c) shows the transformed graph of the example in Figure 1(b)). From now on, by *table graph* we mean the transformed graph.

3.2 Conversion of functions to circuits

Our construction (and, more specifically, FHE below) uses the encoding of the rhs functions of tables as boolean circuits. Such an encoding has already been done in [22], [23]. [22] 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 [23], and is claimed to be 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.

3.3 Cryptographic notation and definitions

We introduce some basic cryptographic notation and definitions that we are going to use, following the notation in [12].

We use the notation *p.p.t.* as an abbreviation of *probabilistic polynomial-time* when referring to algorithms. 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* iff for every p.p.t. algorithm D,

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

We are going to use deterministic private key encryption (symmetric key encryption) schemes, defined as follows (see, e.g., [24]):

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

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

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

(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 $M = SE.Dec(sk,C)$.*

In order for an encryption to be considered secure, we require *message indistinguishability*, i.e., we require that an adversary must not be able to distinguish between two ciphertexts, even if it chooses the corresponding plaintexts itself. More formally:

Definition 9 (Single message indistinguishability) *A private key encryption scheme $SE = (SE.KeyGen, SE.Enc, SE.Dec)$ is* single message indistinguishable *iff for any security parameter K, for any two messages $M, M' \in \{0,1\}^{m(K)}$, $m(K) > K$, and for any p.p.t. adversary A,*

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

where the probabilities are taken over the (random) key produced by $SE.KeyGen(1^K)$, and the coin tosses of A.

An example of a private key encryption scheme satisfying Definitions 8 and 9 is the block cipher Data Encryption Standard (DES) in [25].

3.3.1 Fully Homomorphic Encryption (FHE)

We present the well-known definition of a powerful cryptographic primitive that we will use heavily in our protocols, namely the *Fully Homomorphic Encryption (FHE)* scheme (see [26] and [12] for the history of these schemes and references). This definition is built as the end-result of a sequence of definitions, which we give next:

Definition 10 (Homomorphic Encryption) *A homomorphic (public-key) encryption scheme HE is a tuple of four polynomial time algorithms $(HE.KeyGen, HE.Enc, HE.Dec, HE.Eval)$ with the following properties:*

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

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

(3) $HE.Dec(hsk,\phi)$ is a deterministic algorithm that takes as input a secret key hsk and a ciphertext ϕ, and outputs a message bit.

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

We will require that $HE.Eval$ satisfies the **compactness** property: For all security parameters K, there exists a polynomial $p(\cdot)$ such that for all input sizes n, for all $\phi_1,...,\phi_n$, and for all C, the output length of $HE.Eval$ is at most $p(n)$ bits long.

Definition 11 (C-homomorphic scheme) *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 *iff for every polynomial $n(\cdot)$, sufficiently large K, circuit $C \in C_n$, and input bit sequence $x_1,...,x_n$, where $n = n(K)$, we have*

$$Pr\left[HE.Dec(hsk,\phi) \neq C(x_1,...,x_n),\ s.t. \begin{array}{l} (hpk,hsk) \leftarrow HE.KeyGen(1^K) \\ \phi_i \leftarrow HE.Enc(hpk,x_i),\ i = 1,\dots,n \\ \phi \leftarrow HE.Eval(hpk,C,\phi_1,\dots,\phi_n) \end{array}\right] \le negl(K), \quad (3)$$

where the probability is over the random bits of $HE.KeyGen$ and $HE.Enc$.

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

Definition 13 (IND-CPA security) *A scheme HE is* IND-CPA secure *iff for any p.p.t. adversary A,*

$$\begin{aligned} |Pr[(hpk,hsk) \leftarrow HE.KeyGen(1^K) :& \\ A(hpk, HE.Enc(hpk,0)) = 1] -& \\ Pr[(hpk,hsk) \leftarrow HE.HeyGen(1^K) :& \\ A(hpk, HE.Enc(hpk,1)) = 1]| \le negl(K).& \end{aligned}$$

Fully homomorphic encryption has been a concept known for a long time, but it was not until recently that Gentry [3] gave a feasible implementation of FHE. Our work is independent of particular FHE implementations; we only require their existence. For simplicity, sometimes we write $FHE.Enc(x)$ when the public key hpk is not needed explicitly. For an m-bit string $x = x_1...x_m$, we write $FHE.Enc(x)$ instead of the concatenation $FHE.Enc(x_1) \odot \ldots \odot FHE.Enc(x_m)$, and we do the same for $FHE.Dec$ as well. Similarly, for $FHE.Eval$ with a circuit C as its input such that C outputs m bits $C_1, C_2, \ldots, C_m$, sometimes we write $FHE.Eval(hpk, C, FHE.Enc(hpk,x))$ to denote the concatenation $FHE.Eval(hpk, C_1, FHE.Enc(hpk,x)) \odot \ldots \odot FHE.Eval(hpk, C_m, FHE.Enc(hpk,x))$.

We usually use $\lambda = \lambda(K)$ to denote the ciphertext length of a one-bit FHE encryption.

Next, we present the *multi-hop homomorphism* definition of [27]. Multi-hop homomorphism is an important property for our algorithms, because it allows using the output of one homomorphic evaluation as an input to another homomorphic evaluation.

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

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

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

HE is a multi-hop homomorphic encryption scheme *iff it is i-hop for any polynomial $i(\cdot)$.*

Not all homomorphic encryption schemes satisfy this property, but [27], [26] show that it holds for fully homomorphic encryption schemes. In our algorithms we will use FHE schemes, that also satisfy the IND-CPA security property of Definition 13.

3.3.2 Bit commitment protocols

Following Naor [4], a Commitment to Many Bits (CMB) protocol is defined as follows:

Definition 15 (Commitment to Many Bits (CMB) Protocol) *A CMB protocol consists of two stages:*

- **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 enter a message exchange, and at the end of the stage Bob has some information Enc_D about D.*
- **The revealing stage:** *Bob knows D at the end of this stage.*

The protocol must satisfy the following property for any p.p.t. Bob, for all polynomials $p(\cdot)$, and for a large enough security parameter K:

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

We are going to use the construction of a CMB protocol by Naor [4].

4 Secure and trusted verification for honest developers

4.1 Construction outline

In this section, we construct a secure verification scheme satisfying Definition 6. In a nutshell, we are looking for a scheme that, given an encrypted table graph and its structure graph, will verify the *correctness* of evaluation on a set of test external inputs, in a *secure* way; to ensure the latter, we will require that the intermediate table inputs/outputs produced during the verification process are also encrypted.

VS.Encrypt: In order to encrypt the rhs functions of the tables as well as the intermediate inputs/outputs, we use *universal circuits* [28]. If we represent each function F_i in (transformed) table $T_i = (True, F_i)$ as a boolean circuit C_i (see Appendix 3.2 for methods to do this), then we construct a universal circuit U and a string S_{C_i} for each C_i, so that for any input x, $U(S_C, x) = C(x)$. Following [29] (a method also used in [12]), S_{C_i} and x can be encrypted, while the computation can still be performed and output an encryption of $C(x)$.

Therefore, $VS.Encrypt$ fully homomorphically encrypts S_{C_i} with FHE's public key hpk to get E_{C_i}, and replaces each table T_i with its encrypted version $T'_i = (True, E_{C_i})$. Then, if a verifier wants to evaluate T'_i at x, it gets the FHE encryption x' of x, and runs $FHE.Eval(hpk, U, E_{C_i}, x')$ to get $T'_i(x') = FHE.Eval(hpk, U, E_{C_i}, x')$. Because $FHE.Eval(hpk, U, E_{C_i}, x') = FHE.Enc(hpk, C(x))$ holds, we have that $T'_i(x') = FHE.Enc(hpk, C(x))$. Note that the encrypted graph G' that $VS.Encrypt$ outputs maintains the same structure graph G_{struc} as the original table graph G, and that $VS.Encrypt$ outputs hpk, U in addition to G'. Algorithm 1 implements $VS.Encrypt$.

VS.Encode: $VS.Encode$ is going to address two cases:

Case 1: Suppose the verifier is evaluating an encrypted table T'_i whose output is an external output. From the construction of T'_i, we know that its output is an FHE-encrypted ciphertext. But the verifier needs the *plaintext* of this ciphertext in order for verification to work. We certainly cannot allow the verifier itself to decrypt this ciphertext, because then the verifier (knowing the secret key of the encryption) would be able to decrypt the encrypted circuit inside T'_i as well. What we can do is to allow the verifier to ask the developer for the plaintext; then the latter calls $VS.Encode$ to decrypt this ciphertext for the verifier.

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 $d$, a corresponding string $S_C$ can be efficiently (in terms of $s$ and $d$) computed from $C$, so that $U(S_C, x) = C(x)$ for any input $x$.
3: Suppose $C$ outputs $m$ bits. Construct $m$ circuit $U_1, ..., U_m$ such that for input $x$ and any $i \in [m]$, $U(x, S_C)$ outputs the $i$th bit of $U(x, S_C)$.

4: for all $T_i \in TS, i \in \{1, ..., n\}$ do
5:     Let $C_i$ be the circuit that $C_i(x) = F_i(x)$
6:     Construct the string $S_{C_i}$ from $C_i$
7:     $E_{C_i} \leftarrow FHE.Enc(hpk, S_{C_i})$
8:     $T_i' \leftarrow (True, E_{C_i})$
9: $TS' \leftarrow \{T_1', ..., T_n'\}$
10: $G'_{struc} \leftarrow G_{struc}$
11: return $G' = [TS', G'_{struc}], hpk, U = (U_1, ..., U_m)$
```

Case 2: Suppose the verifier is evaluating an encrypted table T_i' whose output is an internal output used as an input to another table. In this case, we cannot allow the verifier to simply ask the developer to decrypt this output as before; that would give away the intermediate outputs, which are supposed to be kept secret. At the same time, the verifier must be able to figure out whether the actual (unencrypted) value of this intermediate output is $(\bot, \bot)$ or not, i.e., “meaningful” or not. More specifically, V should be able to tell whether intermediate output $FHE.Enc(\top, b)$ contains $\top$ or $\bot$. (Recall from Section 3 that an original table T_i in G outputs a symbol $\bot$ if the predicate in the initial table graph is not satisfied, i.e., T_i’s output is not “meaningful”.)

In Case 2, particular care must be taken when the verifier chooses an external input $x = (\top, a)$ for table T_i'. The verifier is required to use the FHE encryption x' of x in order to evaluate T_i'. An obvious way to do this is to let the verifier compute $FHE.Enc(hpk, (\top, a))$, and evaluate

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

However, this simple solution can be attacked by a malicious verifier as follows: The verifier chooses one of the intermediate outputs, and claims that this intermediate output is the encryption of the external input. Then Case 1 applies, and the verifier asks the developer for the output of $VS.Encode$. Then it is obvious that through this interaction with the developer, the verifier can extract some partial information about the intermediate output. We use $VS.Encode$ to prevent this malicious behaviour: For any external input chosen by the verifier, the latter cannot fully homomorphically encrypt the input by itself; instead, it must send the input to the developer, who, in turn, generates the FHE encryption by calling $VS.Encode$.

In order to allow $VS.Encode$ to distinguish between the two cases, we introduce an extra input parameter that takes a value from the set of special ’symbols’ $\{q_1, q_2\}$ meaning the following:

q_1: $VS.Encode$ is called with $(i, x^i, null, q_1)$. Index i indicates the ith table $T_i' \in G'$ and x^i is an external input to T_i' (Case 1).

q_2: $VS.Encode$ is called with $(i, x^i, T_i'(x^i), q_2)$. Index i indicates the ith table $T_i' \in G'$ and x^i is an intermediate input to T_i' (Case 2).

Also, we allow $VS.Encode$ to store data in a memory M, which is wiped clean only at the beginning of the protocol. Algorithm 2 implements $VS.Encode$.

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

```
if $q = q_1$ then                                              ▷ Case 1
    if $|u| \neq m$ or first component of $u$ is not $\top$ then ▷ $m$ defined in line 3 of Algorithm 1
        return $null$
    else
        $w \leftarrow FHE.Enc(hpk, u)$
        store $(i, w, null)$ in $M$
    return $w$
if $q = q_2$ then                                              ▷ Case 2
    for all $x_j^i \in u$ do                                   ▷ $u = (x_1^i, x_2^i, \ldots)$
        if $x_j^i$ is an output of some $T_k'$, according to $G'$ then
            if $\nexists(k, x^k, T_k'(x^k)) \in M$ such that $x_j^i = T_k'(x^k)$ then return $null$
            else if $\exists(k, x^k, T_k'(x^k)) \in M$ such that $x_j^i = T_k'(x^k)$ and $T_k'(x^k)$ is an FHE encryption of $(\bot, \bot)$ then return $null$
        else if $x_j^i$ is an external input, according to $G'$ then
            if $\nexists(i, w, null) \in M$ such that $x_j^i = w$ then return $null$
    for all $k \in \{1, ..., m\}$ do
        $s_k = FHE.Eval(hpk, U_k, u, E_{C_i})$                    ▷ recall that $U = (U_1, U_2, \ldots, U_m)$
    if $[s_1 s_2 ... s_m] \neq v$ then return $null$
    else
        store $(i, u, v)$ in $M$
        for all $i \in \{1, ..., m\}$ do
            $b_i \leftarrow FHE.Dec(hsk, s_i)$
        if $v$ is not an external output and $b_1 b_2 ... b_{m/2} \neq \bot$ then return $\top$
        else return $b_{m/2+1} b_2 ... b_m$
```

VS.Eval: $VS.Eval$ is an algorithm that allows anyone to check whether the verification was done correctly. In order to achieve this, the interaction between the verifier and the developer is recorded in a public file QA_E. By reading QA_E, $VS.Eval$ can infer which inputs and tables the verifier evaluates, and what outputs it gets. This allows $VS.Eval$ to evaluate the tables on the same inputs, *using its own verifier*, and *at any time*, and check whether the outputs are the recorded ones. Obviously, by using only an honest verifier and QA_E, $VS.Eval$ essentially checks whether the verifier that interacted with the developer originally was also honest.

More specifically, $VS.Eval$ takes input $(1^K, QA_E)$, and outputs 1 or 0 (i.e., accept or reject the verification, respectively). The record file $QA_E = \{(Q_1,A_1),...,(Q_n,A_n),VGA,s,CP,G',hpk,U\}$ records a sequence of *(verifier query, developer reply)* pairs (Q_i,A_i), where Q_i is the verifier query, and A_i is the reply generated by the developer when it runs $VS.Encode(Q_i)$ (the last pair obviously records the verifier's output). It also records test-generator VGA, the set of critical points CP used, and the random seed s used by VGA to produce the test points. Finally, it records the encrypted table graph G', the public FHE key hpk, and the universal circuit U. As mentioned above, the pairs (Q_i,A_i) are public knowledge. Algorithm 3 implements $VS.Eval$. V' is the hardwired honest verifier.

Algorithm 3 $VS.Eval(1^K, QA_E)$

```
for all $(Q_i,A_i) \in QA_E$ do
    Let $Q_i = (r,u,v,q_2)$
    $T'_r(u) \leftarrow FHE.Eval(hpk,U,E_{C_r},u)$          ▷ recall that
$T'_r = (True,E_{C_r})$ is in $G'$
    if $T'_r(u) \neq v$ then
        return 0
Run honest verifier $V'(1^K,G',G^{spec},VGA(s),CP)$
for all $T'_i(x^i)$ that $V'$ chooses to evaluate do
    if $\nexists(Q_j,A_j) \in QA_E$ with $Q_j = (i,x^i,T'_i(x^i),q_2)$ then
        return 0
if $V'$'s output $\neq$ $V$'s output then
    return 0
return 1
```

VS.Path: The purpose of grey-box verification is to allow testing algorithms like MC/DC, which use the structure of publicly-known table graph G_{struc}, to run without restricting the set of evaluation paths they can test. Therefore, we provide the verifier with the ability to choose and evaluate any path in G_{struc}. The verifier picks a path in the table graph, passes it to the developer, and the latter runs $VS.Path$ in order to generate an external input that, when used, will lead to the evaluation of the tables on the path chosen by the verifier. Algorithm 3 implements $VS.Path$.

Algorithm 4 $VS.Path(T_1,...,T_p)$

```
if $T_1,\dots,T_p$ form a path $P = T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_p$ in $G_{struc}$ then
    Generate external input $X$ to the table graph $G$, so that the eval-
uation of $G(X)$ includes the evaluation of tables $T_1,\dots,T_p$.
    return $X$
else return $null$
```

4.1.1 Encrypted table graph evaluation

The evaluation of an encrypted table graph G' is more complex than the evaluation of an unencrypted table graph G. The evaluation of G' at an external input X (i.e., $G'(X)$) is done by Algorithm 5.

Algorithm 5 $G'(X)$

```
$V$ uses $X$ to set up external inputs for the tables
for all table $T'_i$ chosen in a consistent order do
    for all $x^i_j \in x^i$ do                              ▷ $x^i = (x^i_1,x^i_2,\dots)$
        if $x^i_j$ is an external input then
            if $x^i_j \neq null$ then
                $V$ asks the developer to call
$VS.Encode(i,x^i_j,null,q_1)$
                $x^i_j \leftarrow VS.Encode(i,x^i_j,null,q_1)$          ▷ $x^i$ is being
replaced with its encoding
            else
                $T'_i(x^i) = null$
                break
        else
            $T'_p$ = the table whose output is $x^i_j$
            if $T'_p$'s output is $null$ then
                $T'_i(x^i) = null$
                break
            else if $VS.Encode(p,x^p,T'_p(x^p)) = \perp$ then
                $T'_i(x^i) = null$
                break
            else $x^i_j \leftarrow T'_p(x^p)$              ▷ $T'_p(x^p)$ is already encoded
    if $T'_i(x^i) = null$ then continue
    else
        $T'_i(x^i) \leftarrow FHE.Eval(hpk,U,E_{C_i},x^i)$  ▷ $V$ evaluates $T'_i(x^i)$
        $V$ asks the developer to call $VS.Encode(i,x^i,T'_i(x^i),q_2)$
        $V$ receives $VS.Encode(i,x^i,T'_i(x^i),q_2)$
$Y = \{y_{i_1},...,y_{i_s}\}$ are the external output values
return $Y$
```

A typical set of rounds between the honest developer and the verifier can be seen in Figure 4. Notice how the generic protocol of Figure 2 is actually implemented by our construction.

4.2 Correctness and security

We have described an implementation of $VS.Encrypt$, $VS.Encode$, $VS.Eval$ of Definition 3. Note that QA_E plays the role of $Certificate$ for $VS.Eval$. In this section we prove the compliance of our scheme with Definitions 4 and 5. Recall that FHE is the fully homomorphic encryption scheme introduced in Section 3.3.1.

4.2.1 Correctness

Theorem 1 *The verification scheme VS introduced in Section 4.1 satisfies Definition 4.*

Proof We will need the following lemma:

Lemma 1 *When $VS.Eval$ outputs 1, for any table $T' \in TS'$ and x, the output y claimed by V as $T'(x)$ must be equal to $T'(x)$, i.e., V evaluates every table correctly.*

Proof Given that the verifier V' used by $VS.Eval$ is an honest one, the lemma is obviously true.

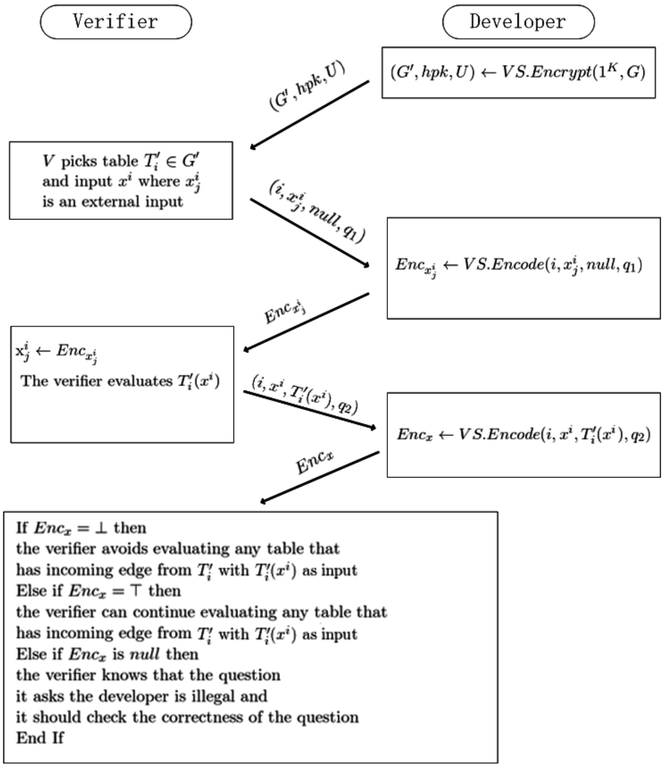

Verifier
Developer
$(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
(G', hpk, U)
V picks table $T'_i \in G'$ and input x^i where x^i_j is an external input
$(i, x^i_j, null, q_1)$
$Enc_{x^i_j} \leftarrow VS.Encode(i, x^i_j, null, q_1)$
$Enc_{x^i_j}$
$x^i_j \leftarrow Enc_{x^i_j}$
The verifier evaluates $T'_i(x^i)$
$(i, x^i, T'_i(x^i), q_2)$
$Enc_x \leftarrow VS.Encode(i, x^i, T'_i(x^i), q_2)$
Enc_x
If $Enc_x = \perp$ then
the verifier avoids evaluating any table that
has incoming edge from T'_i with $T'_i(x^i)$ as input
Else if $Enc_x = \top$ then
the verifier can continue evaluating any table that
has incoming edge from T'_i with $T'_i(x^i)$ 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

Fig. 4: The VS protocol in Section 4.

For the *(query,answer)* pairs in QA_E that do not belong to a consistent traversal of G', Algorithm 2 (lines 11, 12, 14, 17) will return *null* as the corresponding input encodings, and Algorithm 5 (lines 9, 14) will compute *null* for tables with such inputs. Therefore, security and correctness are not an issue in this case. From now on, we will concentrate on the consistent table traversals and their inputs/outputs.

Lemma 2 *Given the external input set X, common to both G' and G, for any table $T_i \in TS$ with input x^i and output $T_i(x^i)$, there is a corresponding table $T'_i \in TS'$ and input w^i, such that*

$$\begin{aligned} w^i &= FHE.Enc(hpk, x^i) \\ T'_i(w^i) &= FHE.Enc(hpk, T_i(x^i)), \end{aligned} \tag{4}$$

where hpk is the public key generated by $VS.Encrypt(1^K, G)$.

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

Base case: Level 1 tables have only external inputs, and, therefore, Algorithm 2 (line 5) returns $w^i = FHE.Enc(hpk, x^i)$. Also, from line 22 of Algorithm 5 we have

$$\begin{aligned} T'_i(w^i) &= FHE.Eval(hpk, U, E_{C_i}, w^i) \\ &= FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i)) \\ &= FHE.Enc(hpk, FHE.Dec(hsk, \\ &\qquad FHE.Eval(hpk, U, E_{C_i}, FHE.Enc(hpk, x^i)))) \\ &= FHE.Enc(hpk, U(S_{C_i}, x^i)) = FHE.Enc(hpk, T_i) \end{aligned}$$

Inductive step: Suppose that for any table $T_i \in TS$ whose level is smaller than $k+1$ and its input x^i, table $T'_i \in TS'$ and its input w^i satisfy (4). Then we show that for any level $k+1$ table $T_j \in TS$ with input $x^j = \{x^j_1, x^j_2, ..., x^j_s\}$, the level $k+1$ table $T'_j \in TS'$ and its *encoded* input w^j satisfy (4).

For any $x^j_p \in x^j$, either $x^j_p = T_{i_p}(x^{i_p})$ for some i_p, or x^j_p is an external input. In case x^j_p is external, then w^j_p satisfies (4) like in the base case. Otherwise, T_{i_p} is a table of level smaller than $k+1$. Then, because of the inductive hypothesis,

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

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

Now, the second part of (4) is proven 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 common external input set X to both G' and G, suppose $T_{i_1}, ..., T_{i_s} \in TS$ are the output level tables that are actually evaluated, and $y_1, \ldots, y_s$ their corresponding outputs. Then Lemma 2 implies that $T'_{i_1}(w^{i_1}) = FHE.Enc(hpk, y_1), \ldots, T'_{i_s}(w^{i_s}) = FHE.Enc(hpk, y_s)$. Accordingly, for every $j \in [s]$, by asking the developer to call $VS.Encode(i_j, w^{i_j}, T'_{i_j}(w^{i_j}), q_2)$, verifier V gets $y_1, ..., y_s$ as the output of Algorithm 2 (line 21).

Lemma 2 (and hence Lemma 3) are based on the verifier evaluating every table correctly and its subgraph traversal being consistent. Assuming that the traversal is indeed consistent (something easy to check on G_{struc}), by running $VS.Eval$, we can know whether V satisfies Lemma 1 ($VS.Eval$ outputs 1). If this is the case, we know that V's evaluation of G' has the same result as the evaluation of G with the same inputs, i.e., (1) in Definition 4 holds. Moreover, if $VS.Eval(1^K, QA_E)$ outputs 1, (2) in Definition 4 also holds (see lines 6 - 12 in Algorithm 3).

4.2.2 Security

Following the approach of [12], we prove the following

Theorem 2 *The verification scheme VS introduced in Section 4.1 satisfies Definition 5.*

Proof In Definition 5 the two simulators S_1 and S_2 simulate $VS.Encrypt$ and $VS.Encode$ respectively. In our implementation we have added $VS.Path$, so we add S_3 to simulate $VS.Path$, together with its own oracle O_2. We can think of (S_2, S_3) as a combination that plays the role of S, and (O_1, O_2) as a combination that plays the role of O in Exp^{ideal} of Figure 3: depending on the kind of query posed by A_2, $S_2^{O_1}$ replies if it is an "Encode" query, and $S_3^{O_2}$ replies if it is a "Path" query. The tuple of simulators (S_1, S_2, S_3) which satisfies Definition 6 is constructed as follows:

- $S_1(1^K, s, d, G_{struc})$ runs in two steps:
  Step 1: S_1 generates its own table graph $\tilde{G} = (\tilde{TS}, G_{struc})$ with its own tables $\tilde{T_i} \in \tilde{TS}$.
  Step 2: S_1 runs $VS.Encrypt(1^K, \tilde{G})$ (Algorithm 1) and obtains G'', hpk, U.
- S_2 receives queries from the A_2 of Definition 5, and can query its oracle O_1 (described in Algorithm 6). $S_2^{O_1}$ actually simulates $VS.Encode$, passing the queries from A_2 to its $VS.Encode$-like oracle O_1. O_1 has a state (or memory) $[M]$ which is initially empty, and contains the mapping of the encrypted table output to the real table output for a given query. S_2 returns the outputs of O_1 as the answers to the queries of A_2.
- $S_3^{O_2}$ simulates $VS.Path$. It receives queries from A_2, and queries its oracle O_2 (described in Algorithm 7) in turn. Then S_2 returns the output of O_2 to A_2.

First, we construct an experiment Exp (see Figure 5), which is slightly different to Exp^{real} in Definition 5. In Exp, the queries of A_2 are not answered by calls to $VS.Encode$ and $VS.Path$, but, instead, they are answered by calls to $S_2^{O_1}$ and $S_3^{O_2}$ (recall that $S_2^{O_1}$ and $S_3^{O_2}$ together are the simulator S_2^{O}). We use the shorthands O'_1 and O'_2 for $S_2^{O_1}$ and $S_3^{O_2}$ in Exp respectively.

$Exp(1^K)$:
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
3. $a \leftarrow A_2^{O'_1, O'_2}(1^K, G', G, CP, hpk, U, state_A)$
4. Output a

Fig. 5: Experiment Exp.

Lemma 4 *Experiments Exp^{real} and Exp are computationally indistinguishable.*

Proof First, note that $VS.Path$ and O'_2 have the same functionality (see Algorithms 4 and 7). We prove that $VS.Encode$ and O'_1 have the same input-output functionality. By the construction of $VS.Encode$ (see lines 1 - 7 in Algorithm 2) and O'_1 (see lines 1 - 7 in Algorithm 6), when a query is of the form (i, u, v, q_1), both algorithms will output the FHE encryption of u, so they have the same input-output functionality. The case of a query (i, u, v, q_2) is more complex; there are two cases:

Algorithm 6 $O_1[M](i, u, v, q)$

```
 1: if $q = q_1$ then                                                   ▷ Case $q_1$
 2:     if $|u| \neq m$ or first component of $u$ is not $\top$ then ▷ $m$ defined in line 3 of Algorithm 1
 3:         return $null$
 4:     else
 5:         $p \leftarrow FHE.Enc(hpk, u)$
 6:         store $(i, (p, null), (u, null))$ in $M$
 7:         return $p$
 8: if $q = q_2$ then                                                   ▷ Case $q_2$
 9:     for all $x_j^i \in u$ do                               ▷ $u = (x_1^i, x_2^i, \ldots)$
10:         if $x_j^i$ is an output of some $T''_k$, according to $G''$ then
11:             if $\exists(k, (y^k, T''_k(y^k)), (z^k, T(z^k))) \in M$ such that $x_j^i = T''_k(y^k)$ then
12:                 $e_j^i \leftarrow T_k(z^k)$
13:                 if $T_k(z^k) = (\bot, \bot)$ then return $null$
14:             else return $null$
15:         else if $x_j^i$ is an external input, according to $G''$ then
16:             if $\exists(i, (y_j^i, null), (z_j^i, null)) \in M$ then
17:                 $e_j^i \leftarrow z_j^i$
18:             else return $null$
19:     for all $i \in \{1, ..., m\}$ do
20:         $s_i = FHE.Eval(hpk, U_i, u, E_{C'_i})$      ▷ recall that $U = (U_1, U_2, \ldots, U_m)$
21:     if $[s_1 s_2 ... s_m] \neq v$ then return $null$
22:     else
23:         $e^i \leftarrow e_1^i e_2^i ... e_m^i$
24:         store $(i, (u, v), (e^i, T_i(e^i)))$ in $M$
25:     if $T_i(e^i) \neq (\bot, \bot)$ and is not an external output then return $\top$
26:     else return the second component of $T_i(e^i)$
```

Algorithm 7 $O_2(T'_1, \ldots, T'_p)$

```
if $T'_1, \ldots, T'_p$ form a path $P = T'_1 \rightarrow T'_2 \rightarrow \ldots \rightarrow T'_p$ then
    Generate an external input $X$ to the table graph $G$ such that the evaluation of $G(X)$ includes the evaluation of tables $T_1, \ldots, T_p$.  ▷ $X$ is generated as in Algorithm 4.
    return X
else return $null$
```

Case 1: Algorithm 2 outputs $null$**.** This happens in the following cases:

1. An intermediate input $x_j^i \in u$ which should be the output of T'_k, is not T'_k's output (see line 11).
2. An intermediate input $x_j^i \in u$ is the output of a T'_k, but T'_k's output is $(\bot, \bot)$ (see line 12).
3. The FHE encryption of an external input $x_j^i \in u$ cannot be found in $VS.Encode$'s memory M, where it should have been if it had already been processed (as it should) by $VS.Encode$ (see line 14).
4. $T'_i(u) \neq v$ (see line 17).

These four cases will also cause Algorithm 6 to output $null$ in lines 10-14 (for the first and second), lines 15 - 18 (for the third), and line 21 (for the fourth).

Case 2: Algorithm 2 doesn't output *null*. Suppose X is the external input to G' and x^i is the input to T_i'. There are three cases for $VS.Encode(i, x^i, T_i'(x^i), q_2)$ (see lines 19 - 23 in Algorithm 2):

1. If T_i''s output is an intermediate output and the FHE encryption of $(\perp, \perp)$, then $VS.Encode$ decrypts $T_i'(x^i)$, and outputs $\perp$.
2. If T_i''s output is an intermediate output and not the FHE encryption of $(\perp, \perp)$, then $VS.Encode$ decrypts $T_i'(x^i)$, and outputs $\top$.
3. If T_i''s output is an external output, then $VS.Encode$ decrypts $T_i'(x^i)$, and outputs the second component of $FHE.Dec(T_i'(x^i))$ (i.e., the actual output).

On the other hand, when $O_1'(i, x^i, T_i'(x^i), q_2)$ calculates $T_i(u^i)$ at u^i (the unencrypted x^i) in lines 8 - 24 of Algorithm 6, there are three cases (see lines 25 - 26 in Algorithm 6):

1. If T_i''s output is an intermediate output and $T_i(u^i)$ is $(\perp, \perp)$, the output is $\perp$.
2. If T_i''s output is an intermediate output and $T_i(u^i)$ is not $(\perp, \perp)$, the output is $\top$.
3. If T_i''s output is an external output, the output is the second component of $T_i(u^i)$.

The first two cases are the same for $VS.Encode(i, x^i, T_i'(x^i), q_2)$ and $O_1'(i, x^i, T_i'(x^i), q_2)$. In the third case, Algorithm 2 outputs the second half of $FHE.Dec(T_i'(x^i))$, while Algorithm 6 outputs the second component of $T_i(u^i)$, which are the same because of (4). Therefore, $VS.Encode$ and O_1' have the same input-output functionality.

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

Lemma 5 *For every p.p.t. adversary $A = (A_1, A_2)$, and any table $T \in G$, consider the following two experiments:*

The outputs of the two experiments are computationally indistinguishable.

Proof If A_2's inputs in $SingleT^{real}$ and $SingleT^{ideal}$ are computationally indistinguishable, then A_2's outputs are also computationally indistinguishable. The former is true, because E_C and $E_{\tilde{C}}$ are two FHE ciphertexts, and, therefore, they are computationally indistinguishable under the IND-CPA security of FHE.

We generate a sequence of $n+1$ different table graphs, each differing with its predecessor and successor only at one table:

$TS^i = (T_1', \ldots, T_{i-1}', T_i', \tilde{T}_{i+1}, \ldots, \tilde{T}_n)$ for $i = 0, 1, \ldots, n$.

All these new table graphs have the same structure graph G_{struc}. To each TS^i, $i \in \{0, \ldots, n\}$ corresponds the experiment Exp^i in Figure 7.

```
    $SingleT^{real}(1^K)$
1.  $(G, CP, state_A) \leftarrow A_1(1^K)$.
2.  $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$
3.  Let $C$ be a circuit that computes $T$'s function.
4.  Generate universal circuit $U = (U_1, \ldots, U_m)$ and string $S_C$, such
    that $U(S_C, x) = C(x)$.
5.  $E_C \leftarrow FHE.Enc(hpk, S_C)$
6.  $a \leftarrow A_2(1^K, (True, E_C), G, CP, hpk, U, state_A)$
7.  Output $a$
```

(a) Experiment $SingleT^{real}$

```
    $SingleT^{ideal}(1^K)$
1.  $(G, CP, state_A) \leftarrow A_1(1^K)$.
2.  $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$
3.  Let $\tilde{G}$ be produced by Step 1 of $S_1(1^K)$
4.  Construct circuit $\tilde{C}$ that computes the function of $\tilde{T} \in \tilde{TS}$.
5.  Generate universal circuit $U = (U_1, ..., U_m)$ and string $S_{\tilde{C}}$, such
    that $U(S_{\tilde{C}}, x) = \tilde{C}(x)$
6.  $E_{\tilde{C}} \leftarrow FHE.Enc(hpk, S_{\tilde{C}})$
7.  $a \leftarrow A_2(1^K, (True, E_{\tilde{C}}), G, CP, hpk, U, state_A)$
8.  Output $a$
```

(b) Experiment $SingleT^{ideal}$

Fig. 6: Experiments $SingleT^{real}$ and $SingleT^{ideal}$.

```
    $Exp^i(1^K)$
1.  $(G, CP, state_A) \leftarrow A_1(1^K)$
2.  $(hpk, hsk) \leftarrow FHE.KeyGen(1^K)$
3.  Generate universal circuit $U = (U_1, \ldots, U_m)$
4.  For $j = 1, \ldots, i$
    • Construct $C_j$ with an $m$-bit output and computes the function
    of $T_j \in G$.
    • Generate string $S_{C_j}$ such that $U(S_{C_j}, x) = C_j(x)$.
    • $E_{C_j} \leftarrow FHE.Enc(hpk, S_{C_j})$
5.  Let $\tilde{G}$ be produced by Step 1 of $S_1(1^K)$
6.  For $j = i+1, ..., n$
    • Construct circuit $\tilde{C}_j$ computing the function of $\tilde{T}_j \in \tilde{G}$.
    • Generate string $S_{\tilde{C}_j}$ such that $U(S_{\tilde{C}_j}, x) = \tilde{C}_j(x)$.
    • $E_{\tilde{C}_j} \leftarrow FHE.Enc(hpk, S_{\tilde{C}_j})$
7.  $G^i \leftarrow [\{(True, E_{C_1}), \ldots, (True, E_{C_i}),$
            $(True, E_{\tilde{C}_{i+1}}), \ldots, (True, E_{\tilde{C}_n})\}, G_{struc}]$
8.  $a \leftarrow A_2^{O_1', O_2'}(1^K, G^i, G, CP, hpk, U, state_A)$
9.  Output $a$
```

Fig. 7: Experiment Exp^i.

Note that Exp^0 is the same experiment as Exp^{ideal}, since Step 5 is the first step of S_1 and Steps 2,3,6,7 are doing exactly what the second step of S_1 does (i.e., $VS.Encrypt$). Also note that Exp^n is the same as Exp in Figure 5, since the $\tilde{G}$ part of Exp^n is ignored, and G^n is the results of $VS.Encrypt(1^K, G)$, i.e., $G^n = G'$.

Now we are ready to prove that Exp^{real} is computationally indistinguishable from Exp^{ideal} by contradiction. Assume that Exp^{real} and Exp^{ideal} are computationally distinguishable, and, therefore, Exp^0 and Exp^n are computationally distinguishable, i.e., there is a pair of p.p.t. adversaries

$A = (A_1, A_2)$ and a p.p.t. algorithm D such that

$$\left|Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]\right| > negl(K). \tag{5}$$

Since

$$\left|Pr[D(Exp^0(1^K)) = 1] - Pr[D(Exp^n(1^K)) = 1]\right| \leq \sum_{i=0}^{n-1} \left|Pr[D(Exp^i(1^K)) = 1] - Pr[D(Exp^{i+1}, (1^K)) = 1]\right|$$

inequality (5) implies that there exists $0 \leq i \leq n-1$ such that

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

We use $A = (A_1, A_2)$ to construct a pair of p.p.t. algorithms $A' = (A'_1, A'_2)$ which together with p.p.t. algorithm D contradict Lemma 5, by distinguishing $SingleT^{real}$ to $SingleT^{ideal}$. Specifically, they determine whether Step 6 of $SingleT^{real}$ or Step 7 of $SingleT^{ideal}$ has been executed. A'_2 can distinguish the two experiments, if it can distinguish between its two potential inputs $(1^K, (True, E_{C_{i+1}}), U, hpk, state_{A'})$ and $(1^K, (True, E_{\tilde{C}_{i+1}}), U, hpk, state_{A'})$. The idea is to extend the table $(True, E_{C_{i+1}})$ or $(True, E_{\tilde{C}_{i+1}})$ (whichever the case) into a full table graph G^i or G^{i+1} (whichever the case), appropriate for experiments Exp^i or Exp^{i+1} (whichever the case), and invoke A_2 which can distinguish between the two. A' is described in Figure 8, where the table graph H is either G^i or G^{i+1}. Hence, by the construction of A', we know that D can be used to distinguish between experiments $SingleT^{real}$ and $SingleT^{ideal}$ for T_{i+1}.

Theorems 1 and 2 imply

Theorem 3 *Our verification scheme satisfies Definition 6.*

5 Secure and trusted verification for general developers

In general, the developer may not comply with Definition 7, i.e., the developer can actually replace $VS.Encode$ with some other malicious algorithm $VS.Encode'$ in its interaction with the verifier. 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 replace our old definition of a verification scheme VS with a new one in Definition 16 below, by adding an algorithm $VS.Checker$, which the verifier can ask the developer to run in order to determine whether the latter indeed runs $VS.Encode$. $VS.Checker$ itself is also run by the developer, which immediately poses the danger of being replaced by some other algorithm $VS.Checker'$. Therefore, $VS.Checker$ must be designed so that even if it is replaced by some other algorithm, the verifier can still figure out that the developer does not run $VS.Checker'$ or $VS.Encode'$ from its replies.

```
$A'_1(1^K)$
1. $(G, CP, state_A) \leftarrow A_1(1^K)$
2. Construct a circuit $C_{i+1}$, and a string $S_{C_{i+1}}$ such that
   $U(S_{C_{i+1}}, x) = C_{i+1}(x)$
3. $E_{C_{i+1}} \leftarrow FHE.Enc(hpk, S_{C_{i+1}})$
4. $T_{i+1} \leftarrow (True, E_{C_{i+1}})$
5. Output $(G, CP, state_{A'})$
```

(a) Algorithm A'_1

```
$A'_2(1^K)$
1. For $j = 1, \dots, i$
   • Construct circuit $C_j$ that computes the function of $T_j \in G$. ($G$
   comes from $A'_1$)
   • Generate string $S_{C_j}$ such that $U(S_{C_j}, x) = C_j(x)$.
   • $E_{C_j} \leftarrow FHE.Enc(hpk, S_{C_j})$
   • Construct table $(True, E_{C_j})$
2. Construct $\tilde{G} = [\tilde{TS}, G_{struc}]$ just as $S_1$ does.
3. For $j = i+2, \dots, n$
   • Construct circuit $\tilde{C}_j$ that computes the function of $\tilde{T}_j \in \tilde{G}$
   • Generate string $S_{\tilde{C}_j}$ such that $U(S_{\tilde{C}_j}, x) = \tilde{C}_j(x)$.
   • $E_{\tilde{C}_j} \leftarrow FHE.Enc(hpk, S_{\tilde{C}_j})$
   • Construct table $(True, E_{\tilde{C}_j})$
4. $H = [\cup_{l=1}^{i}(True, E_{C_l}) \cup \{(True, E_{C_{i+1}})\text{or}(True, E_{\tilde{C}_{i+1}})\}$
       $\cup_{l=i+2}^{n}(True, E_{\tilde{C}_l}), G_{struc}]$
5. $a \leftarrow A_2^{O'_1, O'_2}(1^K, H, G, CP, hpk, U, state_{A'})$
```

(b) Algorithm A'_2

Fig. 8: Algorithms A'_1, A'_2.

In the new extended definition, $VS.Encrypt$, $VS.Encode$, and $VS.Eval$ remain the same. By running $VS.Eval$ with a publicly known $Certificate$, any third party can check whether the developer is malicious and whether the verification was done correctly.

Definition 16 (Extension of Definition 3) *A verification scheme VS is a tuple of p.p.t. algorithms $(VS.Encrypt, VS.Encode, VS.Checker, VS.Eval)$ such that*

- *$VS.Encrypt(1^K, G)$ is a p.p.t. algorithm that takes a security parameter 1^K and a table graph G as input, and outputs an encrypted table graph G'.*
- *$VS.Encode$ is a p.p.t. algorithm that takes an input x and returns an encoding Enc_x.*
- *$VS.Eval$ is a p.p.t. algorithm that takes a security parameter K and a public $Certificate$ as input, has an honest verifier V satisfying Definition 2 hardcoded in it, and outputs 1 if the verification has been done correctly (and 0 otherwise).*
- *$VS.Checker$ is a p.p.t. algorithm with a memory $state_C$ that receives a question Q from the verifier and replies with an answer A, so that the verifier can detect whether the developer indeed runs $VS.Encode$ and $VS.Checker$.*

Figure 9 shows what should be a round of the protocol between the developer and the verifier.

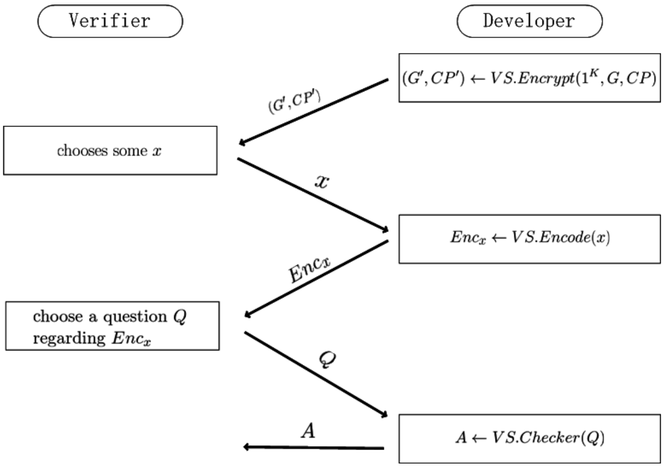

Verifier
Developer
$(G',CP') \leftarrow VS.Encrypt(1^K,G,CP)$
(G',CP')
chooses some x
x
$Enc_x \leftarrow VS.Encode(x)$
Enc_x
choose a question Q regarding Enc_x
Q
$A \leftarrow VS.Checker(Q)$
A

Fig. 9: The generic VS protocol in Definition 16.

Definition 17 (Correctness) *A verification scheme is* correct *iff the following holds:* $VS.Eval(1^K, Certificate) = 1$ *if and only if all of the following hold:*

$$Pr_r[V(1^K,G',G^{spec},VGA_r,CP)(r) = V'(1^K,G',G^{spec},VGA_r,CP))(r)] \geq 1-negl(K), \tag{6}$$

$$\forall x_1,x_2: \; Pr[VS.Encode(x_1) = VS.Encode'(x_1), VS.Checker(x_2) = VS.Checker'(x_2)] \geq 1-negl(K), \tag{7}$$

$$Pr_r[V(1^K,G,G^{spec},VGA_r,CP)(r) = V(1^K,G',G^{spec},VGA_r,CP)(r)] \geq 1-negl(K), \tag{8}$$

where V *is the verifier hardwired in* $VS.Eval$*, and* G' *is the table graph produced by* $VS.Encrypt$*.*

Definition 18 (Security) *For* $A = (A_1,A_2)$ *and* $S = (S_1,S_2,S_3)$ *which are tuples of p.p.t algorithms, consider the two experiments in Figure 10.*

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

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

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

Correspondingly, we update Definition 6:

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

(a) Experiment Exp^{real}

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

(b) Experiment Exp^{ideal}

Fig. 10: The updated experiments.

Definition 19 (Secure and trusted verification scheme) *A verification scheme of Definition 16 is* secure and trusted *iff it satisfies Definitions 17 and 18.*

The updated Definition 7 becomes

Definition 20 *A developer is* honest *iff it always runs* $VS.Encode$ *and* $VS.Checker$*. Otherwise it is called* malicious.

Remark 2 *$VS.Encode$, $VS.Checker$, $S_2^{O_1}$ and $S_3^{O_2}$ in Step 3 of the two experiments in Figure 10 are* not *oracles used by A_2. In these experiments, A_2 plays the role of a (potentially malicious) verifier and interacts with the developer as shown in Figure 9. More specifically, in Exp^{real}, A_2 asks the developer to run $VS.Encode$ or $VS.Checker$ on inputs of its choice (instead of querying an oracle), and receives the answer. In Exp^{ideal}, A_2 again asks the developer, but, unlike Exp^{real}, the latter runs $S_2^{O_1}$ instead of $VS.Encode$ and $S_3^{O_2}$ instead of $VS.Checker$, and provides A_2 with the answer. Hence, whenever we say that A_2 queries $VS.Encode$, $VS.Checker$, $S_2^{O_1}$ or $S_3^{O_2}$ we mean that A_2 asks the developer to run $VS.Encode$, $VS.Checker$, $S_2^{O_1}$ or $S_3^{O_2}$ respectively, and provide the answer. Note that O_1,O_2* are *oracles for S_2,S_3 respectively.*

We repeat that in Section 4 we required that the developer satisfied Definition 5, but that developer may not comply with Definition 20 in this section.

5.1 Construction outline

$VS.Encrypt$, $VS.Encode$ and $VS.Path$ are exactly the same as in Section 4.

VS.Checker: In order to illustrate how $VS.Checker$ is going to be used, we use Figure 11 as an example of the evaluation of a table graph G (on the left) and its encrypted version G' (on the right). There are three potential points where a developer can tinker with $VS.Encode$, namely, when the verifier

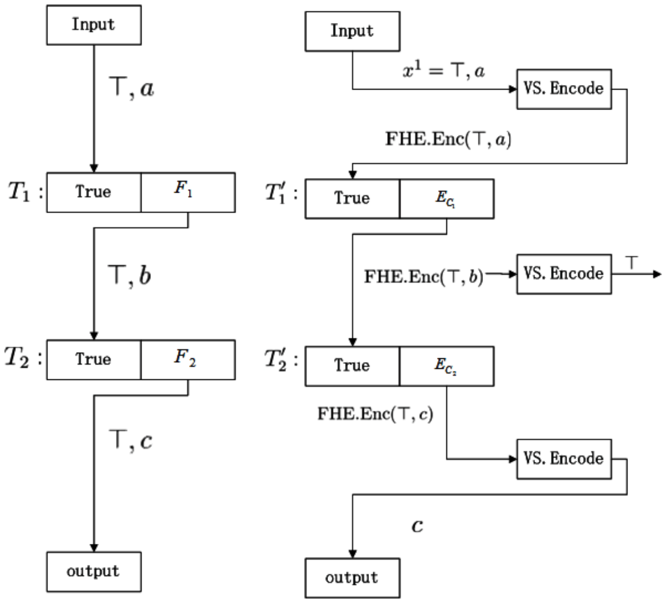


Fig. 11: Example of evaluation of table graph G (left) and its encryption G' (right)

V is querying for an external output (bottom application of $VS.Encode$ in Figure 11), for an intermediate output (middle application of $VS.Encode$ in Figure 11), and for an external input (top application of $VS.Encode$ in Figure 11).

Case 1: External output. We start with this case, since it is somewhat more straight-forward; so, for now, we will assume that the developer has indeed run $VS.Encode$ in the previous tables of the path. Suppose V asks the developer to run $VS.Encode$ to decrypt the external output $FHE.Enc(\top,c)$ of table T_2' in Figure 11. The correct output of $VS.Encode$ should be the external output c, but a malicious developer can replace $VS.Encode$ with a $VS.Encode'$ which outputs $c' \neq c$. V can use the following method to detect this behaviour:

Step 1 Ahead of running the protocol, V announces the use of a deterministic private key encryption scheme SE (see Definition 8), and chooses a secret key sk. (SE's encryption algorithm is $SE.Enc$, represented in a circuit format compatible with $FHE.Eval$.)

Step 2 V first extracts the second component of $FHE.Enc(\top,c)$, which is $FHE.Enc(c)$, and then runs $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(c))$ to get a result y.

Step 3 V sends y to the developer and asks it to run $VS.Checker$ in order to fully homomorphically decrypt y; 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 get $SE.Dec(sk,d)$.

Step 5 V compares $SE.Dec(sk,d)$ with the (known) external output c. If they are the same, V knows that the developer indeed run $VS.Encode$, otherwise V knows that the developer is malicious.

Obviously, if the developer decides to run $VS.Checker(y)$, i.e., $VS.Checker(FHE.Enc(SE.Enc(sk,c)))$, it gets $d = FHE.Dec(y) = SE.Enc(sk,c)$ which it returns to V. Then V obtains c by evaluating $SE.Dec(sk,d)$, and compares it with the answer from the developer who is expected to run $VS.Encode$. If they are not the same, V rejects. Now suppose that the developer runs some $VS.Checker'$ instead of $VS.Checker$, sending V a value $d' \neq d$, and some $VS.Encode'$, sending V a value $c' \neq c$. Then V uses its secret key sk to decrypt d' and gets $SE.Dec(sk,d')$, which is not c' whp, leading to V rejecting whp.

Case 2: Intermediate output. Again, assuming (for now) that the developer has indeed run $VS.Encode$ in the previous tables of the path in Figure 11, this is the case of V getting the intermediate output $FHE.Enc(\top,b)$. 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 $\perp$ (cf. Section 4.1). A malicious developer can run $VS.Encode'$ instead of $VS.Encode$ and return $\perp$ instead of $\top$. The method for $VS.Checker$ described in Case 1 can also be used here by changing Step 2, so that V first extracts the first half of $FHE.Enc(\top,c)$ (which is $FHE.Enc(\top)$), and then runs $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), FHE.Enc(\top))$ to get y.

Case 3: External input. This is the case of an external input $(\top,a)$ (see top of Figure 11). The verifier can treat this case exactly like Case 1, to confirm that $FHE.Enc(\top,a)$ is actually a fully homomorphic encryption of $(\top,a)$.

By doing a consistent traversal of G_{struc}, and applying the relevant Case 1-3 each time (i.e., starting with external inputs (Case 3) and working its way through finally the external outputs (Case 1)), V can enforce the developer to run $VS.Encode$ (and $VS.Checker$) in all steps. Unfortunately, allowing V to ask the developer to run $VS.Checker$ actually makes V far more powerful, and there is a risk that V may abuse this power, by sending queries with malicious content to $VS.Checker$. Therefore we need to force V into asking the 'right' queries. For example, in Step 3 of Case 1, $VS.Checker$ must check whether $y = FHE.Enc(SE.Enc(sk,c))$ before it decrypts y. Our solution to this problem is to use a bit commitment protocol during the interaction between the verifier V and the developer. Algorithm 8 implements $VS.Checker$. The definition of QA_E is exactly the same as in Section 4.1.

VS.Eval: $VS.Eval$ is an extension of $VS.Eval$ in Section 4. 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

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

```
if $|y| \neq l \cdot m$ or $|p| \neq l \cdot m$ then
    return null
else if $\nexists(Q_k,A_k) \in QA_E$ such that $Q_k = (i,x^i,T_i'(x^i),q_2)$ and $p = T_i'(x^i)$ then
    return null
else if $\nexists(Qe_k,Ae_k) \in QA_E$ such that $(Qe_k,Ae_k) = ((i,u_j^i,null,q_1),w_j^i)$ and $p = w_j^i$ then
    return null
else
    for all $j \in \{0,...,m-1\}$ do
        $b_{j+1} \leftarrow FHE.Dec(hsk,y[j \cdot l+1:(j+1) \cdot l])$
    The developer starts the bit commitment protocol described in Section 3.3.2. The developer wants to commit the verifier to $d = b_1,...,b_m$.
    if (bit commitment protocol failed) then return null
    if $\exists(Qe_k,Ae_k) \in QA_E$ such that $Qe_k = (i,x^i,T_i'(x^i),q_2)$ and $p = T_i'(x^i)$ and $T_i'$'s output is an intermediate output then
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p[1:(m/2 \cdot l)]) \neq y$ then
            return $null$
    else if $\exists(Qe_k,Ae_k) \in QA_E$ such that $Qe_k = (i,u_j^i,null,q_1)$ and $p = Ae_k$ and $T_i'$'s input is an external input then
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p) \neq y$ then
            return $null$
    else
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p[(l \cdot m/2+1):(l \cdot m)]) \neq y$ then
            return $null$
return $d$
```

the verifier V evaluates the table graph correctly, $VS.Eval$ runs Algorithm 3. In addition, by reading the log file which records the interaction between the verifier and the developer, $VS.Eval$ can check whether the verifier actually asks the developer to run $VS.Checker$ for every answer it gets (allegedly) from $VS.Encode$. Actually, $VS.Eval$ can recreate the whole interaction between the verifier and the developer from this log file.

The log file is QA_E (as before) but extended with an extra set QA_C which contains tuples (Q_i,A_i,S_i); each such tuple contains a query $Q_i = (i,p,y)$ by V to $VS.Checker$, 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 after running $VS.Checker$. Hence, for each $VS.Encode$ record $(Qe_i,Ae_i) \in QA_E$ generated, a $VS.Checker$ record $(Qc_i,Ac_i,Sc_i) \in QA_C$ will also be generated.

Algorithm 9 implements $VS.Eval$, taking logs QA_E,QA_C as its input.

A typical set of rounds between the developer and the verifier can be seen in Figure 12. Notice how the generic protocol of Figure 9 is actually implemented by our construction.

Algorithm 9 $VS.Eval(1^K,QA_E,QA_C)$

```
if Algorithm 3 returns 0 then    ▷ Algorithm 3 is the old $VS.Eval$ (Section 4)
    return 0
for all $(Qe_i,Ae_i) \in QA_E$ do
    if $\nexists(Qc_i,Ac_i,Sci) \in QA_C$ corresponding to $(Qe_i,Ae_i)$ then return 0
    else find the corresponding $((i,p,y),d,Sc_i) \in QA_C$
    Use hardwired honest verifier $V$ and information $Sc_i$ to replay the bit commitment protocol, and check whether $d$ produced by $VS.Checker$ is the value $d'$ the original verifier committed to.
    if $d' \neq d$ then return 0
    if $Qe_i$'s format is $(i,v_j^i,null,q_1)$ then
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p) \neq y$ then return 0
        else if $SE.Dec(sk,d) \neq v_j^i$ then return 0
        else return 1
    else if $Qe_i$'s format is $(i,x^i,T_i'(x^i),q_2)$ and $T_i'(x^i)$ is an external output then
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p[(l \cdot m/2+1):(l \cdot m)]) \neq y$ then return 0
        else if $SE.Dec(sk,d) \neq Ae_i$ then return 0
        else return 1
    else if $Qe_i$'s format is $(i,x^i,T_i'(x^i),q_2)$ and $T_i'(x^i)$ is an intermediate output then
        if $FHE.Eval(hpk,SE.Enc,FHE.Enc(sk),p[1:(l \cdot m/2)]) \neq y$ then return 0
        else if $SE.Dec(sk,d) \neq Ae_i$ then return 0
        else return 1
```

5.2 Correctness and security

In our implementation, the $Cerificate$ used by $VS.Eval$ in Definition 16 is (QA_E,QA_C,G',hpk,U). In Definition 18, simulators S_1, S_2 and S_3 simulate $VS.Encrypt$, $VS.Encode$ and $VS.Checker$ respectively. Similarly to Section 4.2, we add one more simulator S_4 to simulate $VS.Path$, and its corresponding oracle (Algorithm 7). Algorithm 10 describes the oracle O_3 used by S_4 to simulate $VS.Checker$.

5.2.1 Correctness

We show the following

Theorem 4 *The verification scheme VS introduced in this section satisfies Definition 17.*

Proof We prove that if $VS.Eval(1^k,QA_E,QA_C) = 1$, then inequalities (6)-(8) hold.

Lemma 6 *If* $VS.Eval(1^k,QA_E,QA_C) = 1$, *then* (7) *holds.*

Proof Suppose that $Qc_i = (k,p,y)$ and $Ac_i = d$. For brevity reasons, we will only consider the case of $Qe_i = (k,x^k,T_k'(x^k),q_2)$, and $T_k'(x^k)$ is an external output. The other cases ($Qe_i = (k,x_j^k,null,q_1)$ or $T_k'(x^k)$ is not an external output), are similar. Also, since the bit commitment protocol succeeds whp, V can be assumed to be honest.

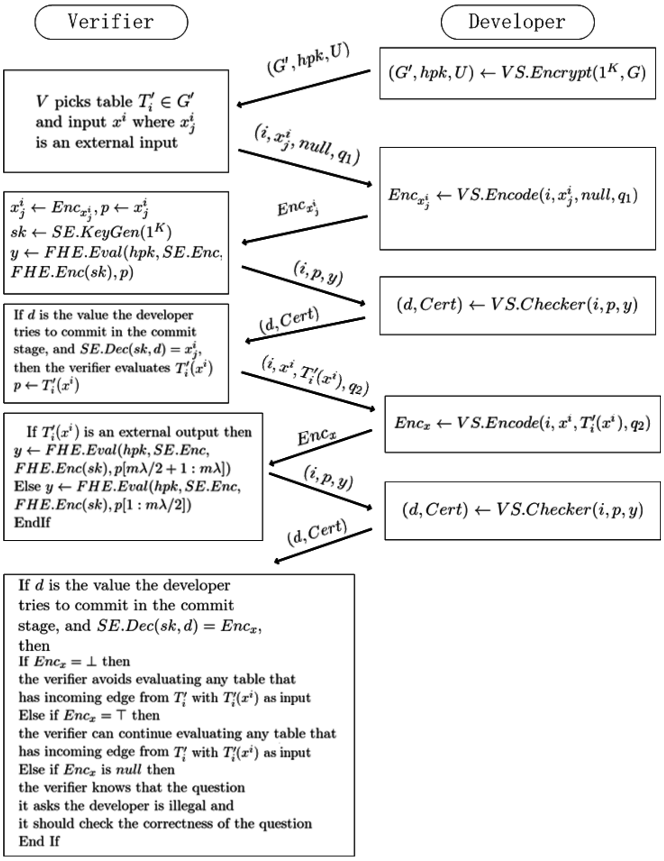

Verifier
Developer
(G', hpk, U)
$(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$
V picks table $T'_i \in G'$ and input x^i where x^i_j is an external input
$(i, x^i_j, null, q_1)$
$Enc_{x^i_j} \leftarrow VS.Encode(i, x^i_j, null, q_1)$
$Enc_{x^i_j}$
$x^i_j \leftarrow Enc_{x^i_j}, p \leftarrow x^i_j$
$sk \leftarrow SE.KeyGen(1^K)$
$y \leftarrow FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p)$
(i, p, y)
$(d, Cert) \leftarrow VS.Checker(i, p, y)$
$(d, Cert)$
If d is the value the developer tries to commit in the commit stage, and $SE.Dec(sk, d) = x^i_j$, then the verifier evaluates $T'_i(x^i)$
$p \leftarrow T'_i(x^i)$
$(i, x^i, T'_i(x^i), q_2)$
$Enc_x \leftarrow VS.Encode(i, x^i, T'_i(x^i), q_2)$
Enc_x
If $T'_i(x^i)$ 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
(i, p, y)
$(d, Cert) \leftarrow VS.Checker(i, p, y)$
$(d, Cert)$
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 T'_i with $T'_i(x^i)$ as input
Else if $Enc_x = \top$ then
the verifier can continue evaluating any table that has incoming edge from T'_i with $T'_i(x^i)$ 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

Fig. 12: The protocol of Section 5.

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

```
if $|y| \neq l \cdot m$ or $|p| \neq l \cdot m$ then
    return $null$
else if $\nexists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, x^i, T_i'(x^i), q_2)$ and $p = T_i'(x^i)$ then
    return null
else if $\nexists(Qe_k, Ae_k) \in QA_E$ such that $(Qe_k, Ae_k) = ((i, u^i_j, null, q_1), w^i_j)$ and $p = w^i_j$ then
    return $null$
else
    Find $(Qe_k, Ae_k) \in QA_E$ that matches the input $(i, p, y)$.
    if $Qe_k$ is $(i, u^i_j, null, q_1)$ then
        $a \leftarrow u^i_j$.
    else $a \leftarrow Ae_k$.
    $b_1 b_2 \ldots b_m \leftarrow SE.Enc(sk, a)$    ▷ Oracle $O_3$ knows secret key $sk$ and $SE.Enc$ used by $A$
    The developer starts the bit commitment protocol described in Section 3.3.2. The developer wants to commit the verifier to $d = b_1, \ldots, b_m$.
    if (bit commitment protocol failed) then return null
    if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_t = (i, x^i, T_i'(x^i), q_2)$ and $p = T_i'(x^i)$ and $T_i'$'s output is an intermediate output then
        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[1 : (m/2 \cdot l)]) \neq y$ then
            return $null$
    else if $\exists(Qe_k, Ae_k) \in QA_E$ such that $Qe_k = (i, u^i_j, null, q_1)$ and $Ae_k = p$ and $T_i'$'s input is an external input then
        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p) \neq y$ then
            return $null$
    else
        if $FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), p[(l \cdot m/2 + 1) : (l \cdot m)]) \neq y$ then
            return $null$
return $d$
```

First we consider the case of $VS.Encode$ and $VS.Encode'$ having the same input-output functionality. Let $(Qc_i, Ac_i, Sc_i) \in QA_C$ be the tuple that corresponds to the current $VS.Encode$ query (Qe_i, Ae_i). Suppose that $VS.Checker'$ outputs a value d, while $VS.Checker$ would output d^*. If $VS.Eval(1^k, QA_E, QA_C) = 1$, then we know (see line 18 in Algorithm 9)

$$SE.Dec(sk, d) = Ae_i \tag{9}$$

Therefore, to prove that $d = d^*$, it is enough to prove that

$$SE.Dec(sk, d^*) = Ae_i. \tag{10}$$

According to $VS.Checker$'s construction (see lines 8-9 in Algorithm 8),

$$d^* = FHE.Dec(hsk, y), \tag{11}$$

where $Qc_i = (i, p, y)$. Since $VS.Encode$ and $VS.Encode'$ have the same input-output functionality, given $Qe_i = (k, x^k, T_k'(x^k), q_2)$ as input to both $VS.Encode$ and $VS.Encode'$, their outputs are the same, i.e., $Ae_i^* = Ae_i$, 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 19-23 in Algorithm 2) ,

$$Ae_i^* = FHE.Dec(hsk, T_k'(x^k)[lm/2 + 1 : lm]) \tag{12}$$

Therefore,

$$Ae_i = FHE.Dec(hsk, T_k'(x^k)[lm/2 + 1 : lm]). \tag{13}$$

On the other hand, whp Definition 11 implies that

$$\begin{aligned} y &= FHE.Eval(hpk, SE.Enc, FHE.Enc(sk), T_k'(x^k)[\frac{lm}{2} + 1 : lm]) \\ &= FHE.Enc(hpk, SE.Enc(sk, FHE.Dec(hsk, T_k'(x^k)[\frac{lm}{2} + 1 : lm])) \end{aligned} \tag{14}$$

Hence, by combining (11) and (14) we get

$$d^* = SE.Enc(sk, FHE.Dec(hsk, T_k'(x^k)[lm/2 + 1 : lm]))),$$

which implies that

$$SE.Dec(sk, d^*) =$$
$$SE.Dec(sk, SE.Enc(sk, FHE.Dec(hsk, T_k'(x^k)[lm/2 + 1 : lm]))) =$$
$$FHE.Dec(hsk, T_k'(x^k)[lm/2 + 1 : lm])$$

(15)

Then, by combining (13) and (15), (10) holds whp.

Next we consider the case of $VS.Checker$ and $VS.Checker'$ having the same input-output functionality. For $(Qe_i,Ae_i) \in QA_E$, and the corresponding pair $(Qc_i,Ac_i) \in QA_C$, $Qe_i = (k,x^k,T'_k(x^k),q_2)$ is the input of $VS.Encode'$ and Ae_i its output, while $Qc_i = (i,p,y)$ is the input of $VS.Checker'$ and $Ac_i = d$ its output. Since $VS.Eval(1^k,QA_E,QA_C) = 1$, we know that (9) holds (see line 18 in Algorithm 9). Also (12) is obviously true, as is (11) (see lines 8-9 in Algorithm 8). The latter, together with the identical functionality of $VS.Checker, VS.Checker'$, implies that

$$d = FHE.Dec(hsk,y) \qquad (16)$$

Hence, by combining (9),(16),(14) we have

$$\begin{aligned}
Ae_i &= SE.Dec(sk,d)\\
&= SE.Dec(sk,FHE.Dec(hsk,y))\\
&= SE.Dec(sk,SE.Enc(sk,FHE.Dec(hsk,T'_k(x^k)[\frac{lm}{2}+1:lm]))\\
&= FHE.Dec(hsk,T'_k(x^k)[lm/2+1:lm])
\end{aligned}$$

Hence, (13) holds, and combined with (13) and (12), we get $Ae_i = Ae^*_i$, i.e., given Qe_i as input to $VS.Encode$ and $VS.Encode'$, their outputs are the same whp.

Finally suppose that there exists $(Qe_i,Ae_i) \in QA_E$ and corresponding $(Qc_i,Ac_i,Sc_i) \in QA_C$ such that $VS.Checker(Qc_i) \neq VS.Checker'(Qc_i)$ and $VS.Encode(Qe_i) \neq VS.Encode'(Qe_i)$. We know (see lines 19-23 in Algorithm 2) that (12) holds, and, by combining (11), (12) and (14), we get

$$\begin{aligned}
&SE.Dec(sk,d^*) = SE.Dec(sk,FHE.Dec(hsk,y))\\
&= SE.Dec(sk,SE.Enc(sk,FHE.Dec(hsk,T'_k(x^k)[\frac{lm}{2}+1:lm])))\\
&= FHE.Dec(hsk,T'_k(x^k)[lm/2+1:lm]) = Ae^*_i
\end{aligned}$$

Since $d^* \neq d$ and $Ae^*_i \neq Ae_i$, d and Ae_i do not satisfy (9) whp. But according to $VS.Eval$'s construction (see line 18 in Algorithm 9), when $VS.Eval(1^k,QA_E,QA_C) = 1$, (9) holds whp, a contradiction.

Lemma 7 *If $VS.Eval(1^k,QA_E,QA_C) = 1$, then (6) and (8) hold.*

Proof Since according to $VS.Eval$'s construction (see line 1 in Algorithm 9), Algorithm 9 outputting 1 implies Algorithm 3 outputting 1, (1) and (2) must hold, and they continue to hold while Algorithm 9 invokes $VS.Checker$. Together with (7) (which we have already proven), (6) and (8) easily follow.

5.2.2 Security

Following the methodology of [12], we will show the following

Theorem 5 *The verification scheme VS introduced in this section satisfies Definition 18.*

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

In our proof, we will need to define a new experiment $Exp^{extra}(1^K)$ (cf. Figure 13). $Exp^{extra}(1^K)$ and $Exp^{real}(1^K)$ (cf. Figure 10) differ only in Step 3, where A_2 queries $VS.Checker$ in $Exp^{real}(1^K)$ and $S_4^{O_3}$ in $Exp^{extra}(1^K)$. If we can show that $VS.Checker$ and $S_4^{O_3}$ have the same input-output functionality, then $Exp^{real}(1^K)$ and $Exp^{extra}(1^K)$ are computationally indistinguishable.

```
    $Exp^{extra}(1^K)$
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 $a$
```

Fig. 13: Experiment Exp^{extra}.

To prove this we distinguish two cases for an input (i,p,y) to $VS.Checker$ and O_3.

First, when $VS.Checker(i,p,y)$ outputs $null$, it is easy to see that $O_3(i,p,y)$ will also output $null$; this happens when the size of p or y is not correct (line 1 in Algorithm 8 and 5 in Algorithm 10), when p is not generated by the evaluation of a table or $VS.Encode$ (line 5 in Algorithm 8 and line 14 in Algorithm 10), and when the bit commitment protocol fails (lines 11-20 in Algorithm 8 and lines 14-23 in Algorithm 10).

Second, when $VS.Checker(i,p,y)$ does not output $null$, it will output a value d; in this case $O_3(i,p,y)$ first generates the same d (see lines 8-12 of Algorithm 10), and after passing the bit commitment protocol, it also outputs d.

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:

$$\begin{aligned}
|Pr[D(Exp^{real}(1^K),1^K) = 1] - Pr[D(Exp^{extra}(1^K),1^K) = 1]|\\
\leq negl(K).
\end{aligned} \qquad (17)$$

$Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ in Figure 14 are the experiments $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ in Definition 5,

```
$Exp^{rtest}(1^K)$
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 $a$
```

(a) Experiment $Exp^{rtest}(1^K)$

```
$Exp^{itest}(1^K)$
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 $a$
```

(b) Experiment $Exp^{itest}(1^K)$

Fig. 14: Experiments Exp^{rtest} and Exp^{itest}.

with $VS.Path$ added to $Exp^{real}(1^K)$ and $S_3^{O_2}$ added to $Exp^{ideal}(1^K)$) (see Figure 3).

In the same way as in Theorem 2, we can show that $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ are computationally indistinguishable, i.e., for all pairs of p.p.t. adversaries $A = (A_1, A_2)$ and any p.p.t. algorithm D,

$$|Pr[D(Exp^{rtest}(1^K), 1^K) = 1] - Pr[D(Exp^{itest}(1^K), 1^K) = 1]| \\ \leq negl(K). \tag{18}$$

Assume that $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally distinguishable. Then (17) implies that $Exp^{extra}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally distinguishable, i.e., there are a p.p.t. algorithm $\tilde{D}$ and a p.p.t. adversary $\tilde{A} = (\tilde{A}_1, \tilde{A}_2)$ such that

$$|Pr[\tilde{D}(Exp^{ideal}(1^K), 1^K) = 1] - Pr[\tilde{D}(Exp^{extra}(1^K), 1^K) = 1]| \\ > negl(K). \tag{19}$$

A p.p.t. adversary $A = (A_1, A_2)$ that wants to distinguish between $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ can use $\tilde{A}$ and $\tilde{D}$ as follows:

- A_1 runs $\tilde{A}_1$ and outputs $(G, CP, state_{\tilde{A}}) \leftarrow \tilde{A}_1(1^K)$.
- A_2 gets one of the two:
  - $(G', hpk, U) \leftarrow VS.Encrypt(1^K, G)$ and has oracle access to $VS.Encode$, or
  - $(G'', hpk, U) \leftarrow S_1(1^K, s, d, G_{struc})$ and has oracle access to $S_2^{O_1}, S_3^{O_2}$

  depending on which one of the two experiments ($Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$) is executed. Then A_2 constructs $S_4^{O_3}$ and $VS.Eval$ (which are also used in $Exp^{ideal}(1^K)$ and $Exp^{extra}(1^K)$), and runs $\tilde{A}_2$ providing it with access to $S_4^{O_3}$. Its output a will be either $\tilde{A}_2^{VS.Encode, VS.Path, S_4^{O_3}}(VS.Eval, G', G, CP, hpk, U, state_{\tilde{A}})$ or $\tilde{A}_2^{S_2^{O_1}, S_3^{O_2}, S_4^{O_3}}(VS.Eval, G'', G, CP, hpk, U, state_{\tilde{A}})$, depending again on which of $Exp^{rtest}(1^K)$ and $Exp^{itest}(1^K)$ is being executed.

If $Exp^{rtest}(1^K)$ is being executed, then it can easily be seen that

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

If $Exp^{itest}(1^K)$ is being executed, then it can easily be seen that

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

But then, (19) implies that

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

which contradicts (18). Therefore, $Exp^{real}(1^K)$ and $Exp^{ideal}(1^K)$ are computationally indistinguishable.

6 Open problems

We have presented protocols that implement secure and trusted verification of a design, taking advantage of any extra information about the structure of the design component interconnections that may be available. Although we show the feasibility of such verification schemes, ours is but a first step, that leaves many questions open for future research.

- **Improving efficiency** Our implementation uses FHE, which, up to the present, has been rather far from being implemented in a computationally efficient way (cf. [30] for a survey of homomorphic encryption implementations, and the references therein). On the other hand, garbled circuits are usually considered to be more efficient than FHE schemes; for example, [23] shows that garbled circuits were much more efficient than a homomorphic encryption scheme in certain Hamming distance computations. Therefore, pursuing protocols based on Yao's garbled circuits is a worthy goal, even if a more efficient garbled circuits construction is less secure.
- **Verifiable computing** Although verifiable computing is not yet applicable to our case (as mentioned in the Introduction), coming up with a method to hide the computation would provide a more efficient solution to the problem of secure and trusted verification, since the amount of re-computation of results needed would be significantly reduced.
- **Hiding the graph structure** Our work has been based on the assumption that the table graph G_{struc} of a design is known. But even this may be a piece of information that the designer is unwilling to provide, since it could

still leak some information about the design. For example, suppose that the design uses an off-the-shelf subdesign whose component structure is publicly known; then, by looking for this subgraph inside G_{struc}, someone can figure out whether this subdesign has been used or not. In this case, methods of hiding the graph structure, by, e.g., node anonymization such as in [31], [32], may be possible to be combined with our or other methods, to provide more security.

- **Public information vs. testing** The extra information we require in order to allow some grey-box test case generation by the verifier, namely the table graph structure, is tailored on specific testing algorithms (such as MC/DC [2]), which produce computation paths in that graph. But since there are other possibilities for test case generation, the obvious problem is to identify the partial information needed for applying these test generation algorithms, and the development of protocols for secure and trusted verification in these cases.
- **Acyclic table graph** As mentioned earlier, we assume that the table graph of the program is acyclic. Lifting of this assumption is currently an open problem.

References

1. T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, Software requirements for the A-7E aircraft. Tech. rep., DTIC Document (1992)
2. K.J. Hayhurst, D.S. Veerhusen, J.J. Chilenski, L.K. Rierson, A practical tutorial on modified condition/decision coverage. Tech. Rep. TM-2001-210876, NASA (2001)
3. C. Gentry, A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)
4. M. Naor, Journal of Cryptology **4**(2), 151 (1991)
5. C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand (1997)
6. C. Wang, J. Hill, J. Knight, J. Davidson, Software tamper resistance: Obstructing static analysis of programs. Tech. Rep. CS-2000-12, University of Virginia (2000)
7. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, in *CRYPTO 2001* (Springer, 2001), pp. 1–18
8. S. Chaki, C. Schallhart, H. Veith, ACM Trans. on Software Eng. and Methodology (TOSEM) **22** (2013)
9. B. Lynn, M. Prabhakaran, A. Sahai, in *EUROCRYPT 2004* (Springer, 2004), pp. 20–39
10. A.C. Yao, in *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE, 1982), pp. 160–164
11. S. Goldwasser, Y.T. Kalai, G.N. Rothblum, in *CRYPTO 2008* (Springer, 2008), pp. 39–56
12. S. Goldwasser, Y. Kalai, R.A. Popa, V. Vaikuntanathan, N. Zeldovich, in *Proceedings of the 45th ACM Symposium on Theory Of Computing (STOC)* (ACM, 2013), pp. 555–564
13. G. Cormode, M. Mitzenmacher, J. Thaler, in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (ACM, 2012), pp. 90–112
14. J. Thaler, M. Roberts, M. Mitzenmacher, H. Pfister, in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing, HotCloud'12* (2012)
15. V. Vu, S. Setty, A.J. Blumberg, M. Walfish, in *Proceeding of the 2013 IEEE Symposium on Security and Privacy, SP'13* (IEEE, 2013), pp. 223–237
16. S. Setty, R. McPherson, A.J. Blumberg, M. Walfish, in *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (2012)
17. S. Setty, V. Vu, N. Panpalia, B. Braun, A.J. Blumberg, M. Walfish, in *Proceedings of the 21st USENIX conference on Security symposium, Security'12* (2012), pp. 253–268
18. B. Parno, J. Howell, C. Gentry, M. Raykova, in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)* (IEEE, 2013), pp. 238–252
19. E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, M. Virza, in *Proceedings of CRYPTO 2013* (Springer, 2013), pp. 90–108
20. S. Arora, S. Safra, Journal of the ACM (JACM) **45**(1), 70 (1998)
21. A. Wassyng, R. Janicki, Fundamenta Informaticae **67**, 343 (2005)
22. D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, in *Proceedings of the 13th conference on USENIX Security Symposium SSYM'04*, vol. 13 (2004), vol. 13, pp. 20–20
23. Y. Huang, D. Evans, J. Katz, L. Malka, in *Proceedings of the 20th USENIX conference on Security SEC'11* (2011), pp. 35–35
24. J. Katz, Y. Lindell, *Introduction to modern cryptography* (CRC Press, 2014)
25. FIPS, NBS **46** (1977)
26. V. Vaikuntanathan, in *Proceedings of the 52nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE, 2011), pp. 5–16
27. C. Gentry, S. Halevi, V. Vaikuntanathan, in *Proceedings of CRYPTO 2010* (Springer, 2010), pp. 155–172
28. L.G. Valiant, in *Proceedings of the 8th annual ACM Symposium on Theory Of Computing (STOC)* (ACM, 1976), pp. 196–203
29. T. Sander, A. Young, M. Yung, in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE, 1999), pp. 554–566
30. A. Acar, H. Aksu, A.S. Uluagac, C. M., arXiv.org (2017). URL https://arxiv.org/abs/1704.03578
31. B. Zhou, J. Pei, in *Proceedings of the 24th IEEE International Conference on Data Engineering, ICDE'08* (IEEE, 2008), pp. 506–515
32. J. Cheng, A.W. Fu, J. Liu, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (ACM, 2010), pp. 459–470

A An example for Section 4

In this subsection we use the example in Figure 15 to show how to apply our verification scheme to actually verify a specific table graph. In Figure 15 there is an initial table graph that is to be verified by the verifier V.

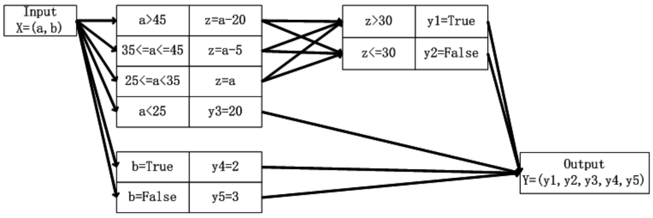


Fig. 15: An initial table graph $\mathcal{G}$ of an implementation

First the developer transforms this initial table graph into a table graph G introduced in Section 2.2 (see Figure 1c). Then

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

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

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

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

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

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

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

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

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 16 is the encrypted table graph G'.

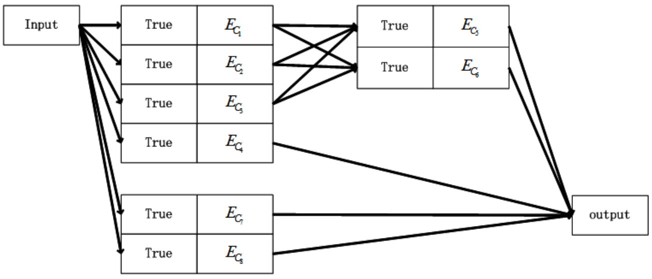


Fig. 16: An encrypted table graph G' after applying VS to G

According to our protocol, the verifier V receives G' and does the verification on G'. We show how V can do MC/DC verification [2] 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 G^{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 $X = (a = 46, b = True)$, V sends the following queries to the developer DL (The queries are in the format of the input of $VS.Encode$): $Q_1 = (1, (\top, 46), null, q_1)$, $Q_2 = (2, (\top, 46), null, q_1)$, $Q_3 = (3, (\top, 46), null, q_1)$, $Q_4 = (4, (\top, 46), null, q_1)$, $Q_5 = (7, (\top, True), null, q_1)$, $Q_6 = (8, (\top, True), null, q_1)$, 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 $\rightarrow PT_1' \rightarrow PT_6' \rightarrow$ 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. $x^6 = PT_1'(x^1)$ is PT_6''s input. After finishing evaluating PT_6', V gets $PT_6'(x^6)$ as the output. Then V sends $(6, x^6, PT_6'(x^6, q_2)$ to DL. DL evaluates $VS.Encode(6, x^6, PT_6'(x^6, q_2)$ and $VS.Encode$'s output is $True$ (Because $(26 < 30)$, PT_5's output is $(\top, False)$. We also know that the output of PT_6' is an external output. Accordingly, $VS.Encode$ outputs $False$). Therefore, DL returns $False$ to V. Then V knows that $y1 = False$ as well as the fact that the lhs predicate (a decision and condition) of $PT_6 \in G$ is satisfied and the rhs function of $PT_6 \in G$ is covered.

After evaluating G' with $X = (a = 46, b = True)$ by similar steps as described above and getting the external output $Y = (\bot, False, \bot, 2, \bot)$, V knows that for X, the lhs predicates of PT_1, PT_6 and PT_7 are satisfied while the rest tables' lhs predicates are not satisfied. Hence, V knows that the predicates of PT_1, PT_6 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_6 and PT_7 are covered. Moreover, V compares Y with $G^{spec}(X)$ 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 G^{spec}.