Differing-Inputs Obfuscation and Applications

Prabhanjan Ananth   Dan Boneh   Sanjam Garg   Amit Sahai   Mark Zhandry

October 24, 2013

Abstract

In this paper we study of the notion of *differing-input obfuscation*, introduced by Barak et al. (CRYPTO 2001, JACM 2012). For any two circuit C_0 and C_1, differing-input obfuscator $\mathsf{di}\mathcal{O}$ guarantees that non-existence of a adversary that can find find an input on which C_0 and C_1 differ implies that $\mathsf{di}\mathcal{O}(C_0)$ and $\mathsf{di}\mathcal{O}(C_1)$ are computationally indistinguishable. We show many applications of this notion:

- We define the notion of differing-input obfuscator for Turing machines and give a construction for the same (without converting it to a circuit) with *input-specific running times*. More specifically, for each input our obfuscated Turning machine takes times proportional to the running time of the Turing machine on that specific input rather than the machines worst-cast running time.
- We give a functional encryption scheme that is *fully-secure* even when the adversary can obtain an unbounded number of secret keys. Furthermore our scheme allows for secret-keys to be associated with Turing machines and thereby achieves *input-specific running times* and can be equipped with *delegation* properties. We stress that no previous scheme in the literature had *any* of these properties.
- We construct the first broadcast encryption system where the ciphertext and secret-key size is constant (i.e. independent of the number of users), and the public key is logarithmic in the number of users. It is the first such scheme where all three parameters are this short.

Both our constructions make inherent use of the power provided by differing-input obfuscation. It is not currently known how to construct systems with these properties from the weaker notion of indistinguishability obfuscation.

1 Introduction

General-purpose program obfuscation aims at making an arbitrary computer programs "unintelligible" while preserving their functionality. The first formal study of the problem of obfuscation was undertaken by Barak et al. in 2001 [BGI+01] where they proposed the notion of *virtual black-box* (VBB) obfuscation. This notion requires that the obfuscation does not leak anything more than what can be learnt with just a *black-box* oracle access to the function. However, unfortunately in the same work Barak et al. showed a family of circuits that cannot be VBB obfuscated.

Weaker variants Obfuscation. In light of this impossibility result Barak et al. left open the problem of realizing weaker notions of obfuscation such as indistinguishability obfuscation and differing-inputs obfuscation (see below for further explanation).

Indistinguishability obfuscation requires that given any two equivalent circuits C_0 and C_1 of similar size, the obfuscations of C_0 and C_1 are computationally indistinguishable. In a very recent work, Garg et al. [GGH$^+$13b], building upon a variant of the multilinear maps framework of Garg et al. [GGH13a], gave the first candidate construction for a general-purpose obfuscator satisfying this notion.

The stronger notion of *differing-inputs* obfuscation states that the existence of an adversary that can distinguish between obfuscations of circuits C_0 and C_1 implies the existence of an adversary that can actually *extract* an input on which the two circuits differ. The starting point for our work is the conjecture that the Garg et al. construction (and variants of it [BR13, BGK$^+$13] indeed achieve the differing-inputs obfuscation notion. Perhaps the strongest evidence for this conjecture is provided by analysis of this construction and variants in suitable generic models [GGH$^+$13b, BR13, BGK$^+$13].[1]

The focus of this paper is to show (1) how to bootstrap the notion of differing inputs obfuscation to build differing inputs obfuscators for Turing Machines with per-input running time, and (2) how to leverage differing inputs obfuscation to obtain a number of interesting applications. Before we turn to our main results, we first illustrate the usefulness of differing inputs obfuscation with two simple examples.

Warmup: Extractable Witness Encryption for NP. As a warmup example we will show how differing-inputs obfuscation can be used to construct extractable witness encryption, a primitive which already has the flavor of extraction. The notion of extractable witness encryption for NP recently introduced in [GGSW13, GKP$^+$13], states that given an NP language L, a extractable witness encryption scheme for L is an encryption scheme that takes as input an instance x and a message bit b, and outputs a ciphertext c. If $x \in L$ and w is a valid witness for x, then a decryptor can use w to decrypt c and recover b. Furthermore security requires that any adversary that can decrypt ciphertext c corresponding to instance x can be used to extract a valid witness for x.

Now we present our construction of extractable witness encryption, which is analogous to the construction of witness encryption from indistinguishability obfuscation as in Garg et al. [GGH$^+$13b]. We define the circuits $C_{x,b}$ for $b \in \{0, 1\}$ taking w as input as follows. If w is a valid witness for x, then $C_{x,b}$ outputs b and $\perp$ otherwise. Differing-input obfuscation of $C_{x,b}$ will serve as an encryption of the bit b. Correctness of decryption is immediate. Recall that security of differing-inputs obfuscation states that existence of an distinguisher between the obfuscations of the two circuits implies the existence of an adversary that can find an input on which the two circuits differ. Thus by the security of differing-inputs obfuscation, we conclude that an adversary breaking the semantic security of the above encryption scheme can be used to extract a valid witness for x.

Example of restricted use software. We find the differing-inputs obfuscation notion stated in its contrapositive form, i.e. non-existence of an adversary that can find input on which the two circuits differ implies the non-existence of an adversary that can distinguish between the obfuscations of the two circuits, as more insightful when considering applications. We highlight how this interpretation can be useful for applications such as *restricted-use software:*

Software developers often want to release multiple tiers of a product with different price points allowing for different levels of functionality. In principle each customer could be provided a separate version of the software enabling only the features he needs. Ideally, a developer could just have flags corresponding to each feature in his software. The developer could then create a customized version of software simply by starting with the full version and then turning off the features the consumer does not want directly at the interface level — requiring minimal additional effort. However, if this is all that is done, then it would be easy for an attacker

[1]In particular, this line of work [GGH$^+$13b, BR13, BGK$^+$13] culminated in an unconditional realization of the VBB notion in the generic multilinear model. The crucial point for our case is that this proof shows extractability of differing inputs from any distinguishing adversary, though only in a generic model.

to bypass these controls and gain access to the full version or the code behind it. The other alternative is for a software development team to carefully remove all unused components — an elaborate task. Can we have the best of both worlds? Our solution is for a developer to release an obfuscated version of the program that takes as input a signature on the custom set of functionality flags that the consumer has paid for. Next we argue that for this application differing-inputs obfuscation suffices. Observe that assuming unforgeability we have that no efficient malicious user can generate a signature on any set of attributes besides the ones provided to it. Given that, differing-inputs obfuscation immediately implies that the obfuscated program with selected features turned off in the perspective of the user is indistinguishable from the obfuscation of the program with unwanted parts removed at the start.

On the other hand, note that indistinguishability obfuscation would not suffice here: This is because the program with the unwanted parts removed implements a different functionality from the original program, and therefore indistinguishability obfuscation alone does not guarantee security in this setting.

1.1 Our Results

We obtain the following results:

Differing-input obfuscation for Turing Machines: We define the notion of differing-input obfuscator for Turing machines and give a construction for the same (without converting it to a circuit) assuming the exitance of differing-input obfuscator for circuits. Moreover our construction achieves *input-specific running times.* This means that evaluating the obfuscated machine on input x does not depend on the worst-case running time of the machine but just on the running time of the unobfuscated machine on input x.

Functional Encryption for Turing Machines: We give a functional encryption scheme that is *fully-secure* even when the adversary can query an unbounded number of secret-key queries. Furthermore our scheme allows for secret-keys to be associated with Turing machines and thereby achieves *input-specific running times* and can be equipped with unbounded *delegation* properties. We stress that no previous scheme in the literature had *any* of these desirable properties.

Short broadcast encryption: In recent work, Boneh and Zhandry [BZ13] show that iO gives a broadcast encryption system with properties that were not previously achievable (see [BZ13] for a survey of related work). While ciphertexts and secret keys in their system are constant size (i.e., independent of the number of users) the size of their public-key is *linear* in the total number of users N. The reason for the linear-size public-key is an obfuscated program used for decryption that takes as input (a representation of) the recipient set $S \subseteq [N]$ and a recipient private key SK_i. The program verifies that recipient i is part of the recipient set S and if so outputs a ciphertext decryption key. Since the recipient set can be linear size, the obfuscated program had to be linear size, thereby forcing the public-key to be linear size. A natural approach to shrink the decryption program in the public-key is as follows: instead of giving the program the recipient set S as an argument, we give it a short proof that i is in S. The obfuscated decryption program will check the proof, and if valid, will decrypt the given ciphertext. A simple proof for the statement $i \in S$ can be built from collision resistant hash functions using Merkle hash trees [Mer88]. Unfortunately, iO is insufficient for proving security of this approach using current techniques. The problem is that using iO we can only puncture a certain PRF embedded in the obfuscated program if the resulting program is *identical* to the original program. However, because the proofs for $i \in S$ are succinct, there *exist* false proofs. That is, for any set $S' \subseteq [N]$ for which $i \notin S'$, there exists a convincing (false) proof that

$i \in S'$. These false proofs prevent us from applying iO to argue that the punctured program is indistinguishable from the original program. While false proofs exist, finding a false proof will break collision resistance of the hash function used to construct the Merkle tree. Therefore, diO can be applied because no polynomial time algorithm can distinguish the punctured program from the original program.

To make this idea work we have to further modify the mechanism used in the broadcast system of [BZ13]. Our final construction is such that proving security requires two applications of diO in three hybrid games. We also show that the same idea can be used to improve the multiparty non-interactive key exchange (NIKE) from [BZ13] so that, even when there is no trusted setup, all parties post at most a logarithmic message (in the number of users) to the public bulletin board. The details are provided in the full version of this paper.

1.2 Related Work.

A concurrent and independent work of [BCP13] also studies differing inputs obfuscation (that they call extractable obfuscation), and obtains a number of applications for differing inputs obfuscation. This work overlaps in part with our own, but [BCP13] also includes results that are not in our work. Most notably, [BCP13] demonstrate a remarkable implication showing that indistinguishability obfuscators *must* satisfy a weak form of differing inputs obfuscation for any pair of circuits that only differ on a polynomial-size set of inputs[2].

2 Preliminaries

2.1 Notation

We represent the security parameter by λ. A function f is said to be negligible in a variable n if for every polynomial p, we have $f(n) < \frac{1}{p(n)}$. For an algorithm A, we use the notation $o \leftarrow A(i)$ to denote that the output of A on input i is o. We use $r \xleftarrow{\$} \mathcal{S}$ to denote that r is drawn from the space $\mathcal{S}$ uniformly at random.

We assume that the reader is familiar with the concept of Turing machines. We denote the running time of Turing machine M on input x by $\mathsf{time}(M, x)$. We say that the output of two Turing machines on an input are the same if the output tapes of the two Turing machines are identical.

For every NP-language L, we associate a corresponding relation R_L such that an instance $x \in L$ iff there exists a witness w such that $(x, w) \in R_L$. Furthermore, we say an instance x is a "valid" (or a true) instance iff $x \in L$. Correspondingly, those instances that don't belong to the language are referred to as invalid (or false) statements.

2.2 Differing-inputs Obfuscation for circuits and TMs

We recall the notion of differing-inputs obfuscation from Barak et. al. [BGI+01]. Next we present this notion for both circuits and Turing machines. Before we go ahead with definition, we describe the notion of differing-inputs circuit family. Intuitively, we call a circuit family to be differing-inputs circuit family if there does not exist any PPT adversary who given two circuits, which are sampled from a distribution defined on this circuit family, can output a value such that both the circuits differ on this input.

Definition 1. *A circuit family $\mathcal{C}$ associated with a sampler* Sampler *is said to be a differing-inputs circuit family if for every PPT adversary $\mathcal{A}$ there exists a negligible function α such*

[2] We note, however, that none of the applications we consider here would work with weak differing inputs obfuscators.

that:

$$\mathsf{Prob}[C_0(x) \neq C_1(x) \ : \ (C_0, C_1, \mathsf{aux}) \overset{\$}{\leftarrow} \mathsf{Sampler}(1^\lambda), \ x \leftarrow \mathcal{A}(1^\lambda, C_0, C_1, \mathsf{aux})] \leq \alpha(\lambda).$$

We now define the notion of differing-inputs obfuscation for a differing-inputs circuit family.

Definition 2. ***(Differing-inputs Obfuscators for circuits)*** *A uniform PPT machine* $\mathsf{di}\mathcal{O}$ *is called a Differing-inputs Obfuscator for a differing-inputs circuit family* $\mathcal{C} = \{C_\lambda\}$ *if the following conditions are satisfied:*

- ***Correctness****: For all security parameters* $\lambda \in \mathbb{N}$*, for all* $C \in \mathcal{C}$*, for all inputs* x*, we have that*
$$\mathsf{Prob}[C'(x) = C(x) \ : \ C' \leftarrow \mathsf{di}\mathcal{O}(\lambda, C)] \ = \ 1$$
- ***Polynomial slowdown****: There exists a universal polynomial* p *such that for any circuit* C*, we have* $|C'| \leq p(|C|)$*, where* $C' = \mathsf{di}\mathcal{O}(\lambda, C)$*.*
- ***Differing-inputs****: For any (not necessarily uniform) PPT distinguisher* D*, there exists a negligible function* α *such that the following holds: For all security parameters* $\lambda \in \mathbb{N}$*, for* $(C_0, C_1, \mathsf{aux}) \overset{\$}{\leftarrow} \mathsf{Sampler}(1^\lambda)$*, we have that*

$$|\mathsf{Prob}[D(\mathsf{di}\mathcal{O}(\lambda, C_0, \mathsf{aux})) = 1] \ - \ \mathsf{Prob}[D(\mathsf{di}\mathcal{O}(\lambda, C_1, \mathsf{aux})) = 1] \leq \alpha(\lambda)$$

The concept of differing-inputs obfuscation can be thought of as a generalisation of indistinguishable obfuscation. This is because, indistinguishable obfuscation is defined for circuits which are identical on all inputs and hence such circuits trivially satisfy the definition of differing-inputs circuit familes. We conjecture that the construction defined in Garg et al. [GGH+13b] and its optimizations from [BR13, BGK+13] satisfies this stronger notion of differing-inputs obfuscation. In this work, using this notion we obtain many applications.

We now consider the case when we are obfuscating Turing machines. Again, to define the differing-inputs property for Turing machines we need to define the notion of differing-inputs Turing machines family. Before we define this, we consider the family of Turing machines $\mathcal{M}$ which is equipped with $\mathsf{Sampler}_{\mathcal{M}}$ which efficiently samples two Turing machines along with auxiliary information. For simplicity, we assume each machine M in the family $\mathcal{M}$ on an input x in addition to its output also outputs the time τ_x it runs in.

Definition 3. *A Turing machine family* $\mathcal{M}$ *associated with a sampler* $\mathsf{Sampler}_{\mathcal{M}}$ *is said to be a differing-inputs machine family if for every PPT adversary, the following holds*

$$\mathsf{Prob}[M_0(x) \neq M_1(x) : (M_0, M_1, \mathsf{aux}) \leftarrow \mathsf{Sampler}(1^\lambda), \ x \leftarrow \mathcal{A}(1^\lambda, M_0, M_1, \mathsf{aux})] = negl(\lambda)$$

Remark 1. Note that for simplicity we have assumed, that all Turing machines in family $\mathcal{M}$ *outputs the time* τ_x *in addition to the output on input* x*. This in particular implies that the two Turing machines output by the sampler on each input* x *run in the same time. We stress that this has been done for simplicity and our definition can also deal with a family of machines with different running times by just padding.*

Similar to the case of circuits, we define the notion of differing-inputs obfuscation for a family of differing-inputs Turing machines.

Definition 4. ***(Differing-inputs Obfuscators for Turing machines)*** *A uniform PPT machine* $\mathsf{di}\mathcal{O}_{\mathsf{TM}}$ *is called a Turing machine Differing-inputs Obfuscator defined for differing-inputs Turing machine family* $\mathcal{M}$*, if the following conditions are satisfied:*

- ***Correctness****: For all security parameters* $\lambda \in \mathbb{N}$*, for all* $M \in \mathcal{M}$*, for all inputs* x*, we have that*
$$\mathsf{Prob}[M'(x) = M(x) \ : \ M' \leftarrow \mathsf{di}\mathcal{O}(\lambda, M)] \ = \ 1$$

- ***Differing-inputs obfuscation property**: For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for $(M_0, M_1, \mathsf{aux}) \xleftarrow{\$} \mathsf{Sampler}_{\mathcal{M}}(1^\lambda)$ we have that*

$$|\mathsf{Prob}[D(\mathsf{di}\mathcal{O}(\lambda, M_0, \mathsf{aux})) = 1] \;-\; \mathsf{Prob}[D(\mathsf{di}\mathcal{O}(\lambda, M_1, \mathsf{aux})) = 1] \leq \alpha(\lambda)$$

*In addition to the above properties if $\mathsf{di}\mathcal{O}_{\mathsf{TM}}$ satisfies the following properties, with respect to a universal polynomial p, then we say that $\mathsf{di}\mathcal{O}_{\mathsf{TM}}$ is **succinct** and has **input-specific run time**.*

- ***Succinct**: The size of M' is $p(\lambda, |M|)$, where $|M|$ denotes the size of the Turing machine M.*
- ***Input-specific run time**: The running time of M' on an input x is $p(\lambda, \mathsf{time}(M, x))$.*

We can also consider the notion of indistinguishability obfuscation for Turing machines. The definition is very similar to the above definition except that the indistinguishability of obfuscations holds only only for Turing machines which are same on all inputs. We present the formal definition in Appendix A for the sake of completeness. We note that our construction of Turing machine differing-inputs obfuscation also satisfies the definition of Turing machine indistinguishable obfuscation since Definition 4 implies Definition 6.

3 Differing-inputs Obfuscators for Turing Machines

In this section, we construct indistinguishability obfuscators for Turing machines. The advantage of considering obfuscation of Turing machines over circuits is two-fold. Firstly, the running time of the obfuscated Turing machine would be input specific. Secondly, the size of the obfuscation does not depend on the worst case running time of the Turing machine. Since the real world applications are programs it is more natural to consider obfuscation of Turing machines rather than circuits.

Our construction is based on the assumption that the differing-input obfuscator for all circuits exists along with the existence of FHE, SNARKs and collision resilient hash functions, all of which are well studied.

3.1 Tools

We now describe the main cryptographic tools that we use in our construction.

Universal Turing machines. Universal Turing machine takes as input a Turing machine, an input on which the Turing machine is executed and a time to indicate the number of steps of execution. The output of the univeral Turing machine is basically the output of the Turing machine on that input if the execution is completed within the time limit, which is given as input to the universal Turing machine. Otherwise, the universal Turing machine outputs $\perp$. We consider a variant of universal Turing machine, that instead of outputting the entire result of execution, will just output one particular bit from the result of execution. More formally, we define the variant as follows. For every $1 \leq i \leq t$, represent by $\mathsf{UTM}_{y,t}^{(i)}(\cdot)$, the following program: It takes as input a Turing machine M' and executes M' on y for t steps. If the execution is completed within t steps then output the i^{th} bit of the output of the execution otherwise output $\perp$.

FHE for Turing machines. Goldwasser et al. in [GKP+13] build a compiler that takes a Turing machine M along with the number of steps t as input and then produces a Turing machine that computes the FHE evaluation of M for t number of steps. In more detail, the compiler converts the machine into an oblivious Turing machine M using the Pippenger-Fischer [PF79] transformation. It then constructs a new Turing machine M_{FHE} which takes a ciphertext along with a FHE public key as input and executes the oblivious Turing machine fully homomorphically on the ciphertext for t number of steps. The output of the compiler is M_{FHE}. The compiler, denoted by $\mathsf{Compile}^{\mathsf{TM}}_{\mathsf{FHE}}$, is described formally in Appendix B.2.

SNARKs. Succinct non interactive arguments of knowledge, referred to as SNARKs, are arguments where the proof sent by the prover to the verifer is succinct. By succinct, we mean that the size of the proof is upper bounded by a fixed polynomial in the security parameter and is independent of the instance for which the proof is given. In addition to succinctness, one other main property satisfied by SNARKs is that the verifier runs in time that depends only on the size of the input instance and the security parameter, and not on the size of the witness. SNARKs have been constructed under knowledge assumptions [BCCT13]. The formal details of SNARKs are presented in Appendix B.3. We denote the SNARK proof system we use by (Setup, P, V).

Hash functions. The final tool we require for our construction are cryptographic hash functions that map arbitrary length input to a fixed length output. More formally, we consider a hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{l(\lambda)}$, where l is a polynomial [3]. There are constructions of such functions known in the literature [Mer90, Dam90, GK03]. Henceforth, we refer to such functions as collision-resilient size reducing hash functions.

3.2 Construction

We are now ready to describe the construction. Before we do this, we first describe a class of programs $\mathcal{P}$, represented by a circuit family to which we apply differing-inputs obfuscation for circuits. Each program in this class is indexed by $(g_1, g_2, \mathsf{CRS}, \mathsf{SK}_1, \mathsf{PK}_1, \mathsf{PK}_2)$. We denote such a program by $\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}_1,\mathsf{PK}_1,\mathsf{PK}_2)}$. Here, $(\mathsf{PK}_1, \mathsf{SK}_1), (\mathsf{SK}_2, \mathsf{PK}_2)$ denotes the FHE public key-secret key pairs, g_1, g_2 denote the encryptions of M with respect to PK_1 and PK_2 respectively and CRS denotes the common reference string output by the SNARK setup algorithm. We describe $\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}_1,\mathsf{PK}_1,\mathsf{PK}_2)}$ in Figure 1.
We are now ready to give the construction of differing-inputs obfuscation of a differing-input Turing machine family.

The obfuscation of a Turing machine is captured by the obfuscate algorithm, denoted by $\mathsf{Obfuscate}_{\mathsf{TM}}$. As in Garg et. al. [GGH+13b], we view the execution of the obfuscated Turing machine on an input as an evaluation algorithm, denoted by $\mathsf{Evaluate}_{\mathsf{TM}}$.

$\mathsf{Obfuscate}_{\mathsf{TM}}(1^\lambda, M)$: The obfuscation algorithm on input a security parameter and a Turing machine M does the following:

1. Generate $(\mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{SK}^1_{\mathsf{FHE}}) \leftarrow \mathsf{Setup}_{\mathsf{FHE}}(1^\lambda)$ and $(\mathsf{PK}^2_{\mathsf{FHE}}, \mathsf{SK}^2_{\mathsf{FHE}}) \leftarrow \mathsf{Setup}_{\mathsf{FHE}}(1^\lambda)$.
2. Generate ciphertexts $g_1 = \mathsf{Encrypt}_{\mathsf{FHE}}(\mathsf{PK}^1_{\mathsf{FHE}}, M)$ and $g_2 = \mathsf{Encrypt}_{\mathsf{FHE}}(\mathsf{PK}^2_{\mathsf{FHE}}, M)$.
3. Compute CRS by executing the setup algorithm Setup corresponding to the SNARK proof system, denoted by (Setup, P, V) for the relation described in Equation 1 in Figure 1.

[3]Technically, this is imprecise. Instead of considering a single hash function, we consider a hash function family from which we sample a hash function $\mathcal{H}$. Whenever we use $\mathcal{H}$, we implicitly mean that $\mathcal{H}$ was sampled from the hash function family.

$$\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}_1,\mathsf{PK}_1,\mathsf{PK}_2)}$$

Given input $(i, e_1^{(i)}, e_2^{(i)}, h_x, t, \varphi)$, $\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK},\mathsf{PK})}$ proceeds as follows:

1. Execute the SNARK verifier V on input $(e_1^{(i)}, e_2^{(i)}, g_1, g_2, h_x, t, i)$ along with CRS and proof φ. The SNARK proof system (P, V) is defined for the language L where $(e_1^{(i)}, e_2^{(i)}, g_1, g_2, h_x, t, i)$ are instances in L with witnesses x such that:

$$M^{(i)}_{\mathsf{FHE}} = \mathsf{Compile}^{\mathsf{TM}}_{\mathsf{FHE}}(\mathsf{UTM}^{(i)}_{x,t}, 2^t \mathsf{log} 2^t) \text{ and } e_1^i = M^{(i)}_{\mathsf{FHE}}(\mathsf{PK}_1, g_1)$$
$$\text{and } e_2^{(i)} = M^{(i)}_{\mathsf{FHE}}(\mathsf{PK}_2, g_2) \text{ and } \mathcal{H}(x) = h_x \tag{1}$$

2. If the verifier rejects then output 0; otherwise, output $\mathsf{Decrypt}_{\mathsf{FHE}}(e_1^{(i)}, \mathsf{SK}_1)$.

Figure 1 A template of a program in the program class $\mathcal{P}$.

4. Generate a differing-inputs obfuscation for the circuit $\mathsf{P1} = \mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}^1_{\mathsf{FHE}},\mathsf{PK}^1_{\mathsf{FHE}},\mathsf{PK}^2_{\mathsf{FHE}})}$ as $\mathsf{P1}_{\mathsf{obf}} = \mathsf{di}\mathcal{O}(\mathsf{P1}, \lambda)$.
5. The output of this algorithm is $\sigma = (\mathsf{P1}_{\mathsf{obf}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}, g_1, g_2, \mathsf{CRS})$.

$\mathsf{Evaluate}_{\mathsf{TM}}(\sigma = (\mathsf{P1}_{\mathsf{obf}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}, g_1, g_2, \mathsf{CRS}), x)$: On input the obfuscation of Turing machine M and input x, the $\mathsf{Evaluate}_{\mathsf{TM}}$ algorithm outputs $M(x)$ as follows.

1. Compute the hash of x using the hash function, $\mathcal{H}$ and denote the result by h_x.
2. Repeat the following for steps $t = 0, 1, \ldots$:
   - Execute $\mathsf{Compile}^{\mathsf{TM}}_{\mathsf{FHE}}(\mathsf{UTM}^{(i)}_{(x,2^t)},\ 2^t \mathsf{log}(2^t))$ [4], for all $1 \le i \le 2^t$, to obtain $M^{(i,t)}_{\mathsf{FHE}}$ [5]. .
   - Compute $e_1^{(i,t)} = M^{(i,t)}_{\mathsf{FHE}}(\mathsf{PK}^1_{\mathsf{FHE}}, g_1)$ and $e_2^{(i)} = M^{(i,t)}_{\mathsf{FHE}}(\mathsf{PK}^2_{\mathsf{FHE}}, g_2)$, where $1 \le i \le 2^t$.
   - For every $1 \le i \le 2^t$, compute SNARK proof φ_i, using prover P, that the encryptions $e_1^{(i,t)}$ and $e_2^{(i,t)}$ as well as the hash value h_x are computed correctly as in Equation 1 in Figure 1.
   - For every $1 \le i \le 2^t$, run $\mathsf{P1}_{\mathsf{obf}}(i, e_1^{(i,t)}, e_2^{(i,t)}, h_x, t, \varphi_i)$. If the output of the program is $\perp$ then go to the beginning of the loop. Else, first assign b_i to be the output of $\mathsf{P1}_{\mathsf{obf}}$. Consider the concatenation of b_i, for $1 \le i \le 2^t$ to be out. The output of $\mathsf{Evaluate}_{\mathsf{TM}}$ is out.

The above construction satisfies both the succinctness as well as the input-specific running time properties. This is proved in Appendix C. We now focus on the correctness as well as the security of the above scheme. Crucial to both these properties is the following lemma which shows that the program class can be implemented by a differing-inputs circuit family. To do this, we first define a PPT algorithm $\mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}$ corresponding to the program class $\mathcal{P}$ as follows. The sampler $\mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}$ receives as input security parameter λ along with $(M_0, M_1, \mathsf{aux}_{\mathcal{M}})$, where $(M_0, M_1, \mathsf{aux}_{\mathcal{M}})$ is the output of $\mathsf{Sampler}_{\mathcal{M}}$, which is the sampler algorithm of $\mathcal{M}$. The sampler $\mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}$ first executes the setup algorithm of FHE twice to obtain

[4]A universal Turing machine that executes an input Turing machine for t steps, itself takes $t\mathsf{log}(t)$ number of steps.

[5]*Note that we need to execute the compile algorithm for every output bit. But, we know that the output length of a Turing machine cannot exceed the running time required to produce that output. And so, we execute the compile algorithm for 2^t number of steps.*

$(\mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{SK}^1_{\mathsf{FHE}}), (\mathsf{PK}^2_{\mathsf{FHE}}, \mathsf{SK}^2_{\mathsf{FHE}})$. Then, the Turing machines M_0 and M_1 are encrypted using public keys $\mathsf{PK}^1_{\mathsf{FHE}}$ and $\mathsf{PK}^2_{\mathsf{FHE}}$ to obtain g_1 and g_2 respectively. Finally, execute the setup algorithm of SNARK proof system to obtain CRS. Output the programs $\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}^1_{\mathsf{FHE}},\mathsf{PK}^1_{\mathsf{FHE}},\mathsf{PK}^2_{\mathsf{FHE}})}$ and $\mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}^2_{\mathsf{FHE}},\mathsf{PK}^1_{\mathsf{FHE}},\mathsf{PK}^2_{\mathsf{FHE}})}$. The auxillary information, denoted by $\mathsf{aux}^{\mathcal{M}}_{\mathcal{P}}$, consists of $(\mathsf{aux}_{\mathcal{M}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}, \mathsf{CRS})$.

Lemma 1. *Consider a class of programs $\mathcal{P}$, defined as before. Let $\mathsf{P1} = \mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}^1_{\mathsf{FHE}},\mathsf{PK}^1_{\mathsf{FHE}},\mathsf{PK}^2_{\mathsf{FHE}})}$ and $\mathsf{P2} = \mathsf{P}^{(g_1,g_2,\mathsf{CRS})}_{(\mathsf{SK}^2_{\mathsf{FHE}},\mathsf{PK}^2_{\mathsf{FHE}}),\mathsf{PK}^2_{\mathsf{FHE}}}$ along with auxillary information aux be the output of sampler algorithm $\mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}$. There does not exist any PPT adversary $\mathcal{A}$ on input $(\mathsf{P1}, \mathsf{P2}, \mathsf{aux})$ outputs y such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$, with non-negligible probability under the assumption that $\mathcal{M}$ is a differing-inputs Turing machine family.*

The proof of the above lemma can be found in Appendix C.1. Using this lemma, we now prove the correctness and the security of the differing-inputs obfuscation scheme.

Correctness. For simplicity, we assume that the obfuscaton of $\mathsf{P1}$, denoted by $\mathsf{P1}_{\mathsf{obf}}$, is executed once, as against multiple times, and the entire output of the FHE evaluation phase is fed to the obfuscation of $\mathsf{P1}$. We now argue the correctness of the scheme. The correctness of the FHE scheme along with the correctness of the SNARK proof system imply that the input to P is an encryption of $M(x)$ followed by a valid proof that the encryption is correctly computed, where x is the input to the obfuscation scheme. Now, note that if a valid encryption of $M(x)$ and a valid proof that $M(x)$ is correctly computed is given to $\mathsf{P1}$ then the output of $\mathsf{P1}$ would be $M(x)$. And so, by the correctness of $\mathsf{di}\mathcal{O}$ it follows that the output of $\mathsf{P1}_{\mathsf{obf}}$ is $M(x)$. This means that the output of the evaluate algorithm is $M(x)$. This proves the correctness of the $\mathsf{di}\mathcal{O}$ scheme.

Security proof. We now describe the security proof of the differing-inputs obfuscation scheme for the Turing machines. The security expermient proceeds by the challenger first executing the sampler algorithm of $\mathcal{M}$ is executed to obtain $(M_0, M_1, \mathsf{aux}_{\mathcal{M}})$. The challenger then sends M_{obf} to the adversary, where M_{obf} is either the $\mathsf{di}\mathcal{O}$ obfuscation of M_0 or M_1. The security guarantee is that the adversary's output when M_0 is obfuscated is negligibly close to its output when M_1 is obfuscated. To show this, we first describe the hybrids which are similar to the security arguments of the indistinguishability obfuscation scheme of the circuits from Garg et al. [GGH+13b]. For completeness sake, we present the hybrids below.

Hybrid_0: This corresponds to the honest execution of the differing-inputs obfuscation corresponding to the Turing machine M_0.

Hybrid_1: In this hybrid, the ciphertext g_1 is generated by encrypting M_0 (under $\mathsf{PK}^1_{\mathsf{FHE}}$) while the ciphertext g_2 is obtained by encrypting the Turing machine M_1 (under $\mathsf{PK}^2_{\mathsf{FHE}}$). The rest of the hybrid is the same as the previous hybrid Hybrid_0.

Hybrid_2: The ciphertexts g_1 and g_2 are generated the same way as in the previous hybrid. The only difference is that instead of obfuscating program $\mathsf{P1}$, the program $\mathsf{P2}$ is obfuscated.

Hybrid_3: In this hybrid, the ciphertexts g_1 is generated by encrypting M_1 (under $\mathsf{PK}^1_{\mathsf{FHE}}$) while the ciphertext g_2 is (still) generated by encrypting M_1 (under $\mathsf{PK}^2_{\mathsf{FHE}}$). As in the previous hybrid, the obfuscation component is still generated from $\mathsf{P2}$.

Hybrid_4: The ciphertexts are generated as in the previous hybrid. That is, g_1 and g_2 are encryptions of M_1 under keys $\mathsf{PK}^1_{\mathsf{FHE}}$ and $\mathsf{PK}^2_{\mathsf{FHE}}$ respectively. But this time, the obfuscation component corresponds to the program $\mathsf{P1}$ instead of $\mathsf{P2}$.

Note that this corresponds to the honest execution of the differing-inputs obfuscation corresponding to the obfuscation of M_1.

We show that every two consecutive hybrids are computationally indistinguishable with respect to each other. The proof of this can be found in Appendix C.2.

4 FE for Turing Machines

We present a construction of functional encryption for Turing machines using the differing-inputs obfuscation construction for Turing machines in Section 3. This is very similar in spirit to the construction of functional encryption in Garg et al. [GGH+13b]. The main tools required for this construction are IND-CPA secure public key encryption scheme (defined in Appendix B.4), simulation sound non interactive zero knowledge (defined in Appendix B.5) and a SNARK proof system (as defined as Appendix B.3). We denote the PKE scheme by $(\mathsf{Setup}_{\mathsf{PKE}}, \mathsf{Encrypt}_{\mathsf{PKE}}, \mathsf{Decrypt}_{\mathsf{PKE}})$, simulation sound NIZK by $(\mathsf{Setup}^{\mathcal{SS}}_{\mathsf{NIZK}}, P^{\mathcal{SS}}_{\mathsf{NIZK}}, V^{\mathcal{SS}}_{\mathsf{NIZK}})$ and a SNARK proof is denoted by $(\mathsf{Setup}_{\mathsf{SNARK}}, P_{\mathsf{SNARK}}, V_{\mathsf{SNARK}})$. Further, we denote the SNARK proof system by $(\mathsf{Setup}_{\mathsf{SNARK}}, P_{\mathsf{SNARK}}, V_{\mathsf{SNARK}})$ The simulation sound NIZK proof is defined for the relation $R_{\mathcal{SS}}$ which is defined later. The SNARK proof system on the other hand is defined for the relation defined in Equation 2.

We describe a class of Turing machines $\mathcal{P}_{\mathsf{FE}}$ which will be useful for our FE construction. Every program in $\mathcal{P}_{\mathsf{FE}}$ is indexed by $(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})$ and we denote such a program by $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$, where f is a function implementable by a Turing machine. Before we describe the program $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$, we specify the relation $R_{\mathcal{SS}}$ for which the SS-NIZK proof system is defined. The relation consists of pairs of ciphertexts $(\mathsf{CT}_1, \mathsf{CT}_2)$ as instances such that both the ciphertexts are the output of the same message m under different public keys $\mathsf{PK}^1_{\mathsf{PKE}}$ and $\mathsf{PK}^2_{\mathsf{PKE}}$. More formally,

$$R_{\mathcal{SS}} = \{(\mathsf{CT}_1, \mathsf{CT}_2; m, r_1, r_2) \,:\, \mathsf{CT}_1 = \mathsf{Encrypt}_{\mathsf{PKE}}(\mathsf{PK}^1_{\mathsf{PKE}}, m; r_1) \text{ and } \mathsf{Encrypt}_{\mathsf{PKE}}(\mathsf{PK}^2_{\mathsf{PKE}}, m; r_2)\}$$

We give the description of the program $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ in Figure 2.

$\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$

Given input $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi, \Pi_{\mathsf{SNARK}})$, $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ proceeds as follows:

1. Execute the SNARK verifier V on input $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi, \mathsf{CRS}_{\mathcal{SS}})$ along with strings $\mathsf{CRS}_{\mathsf{SNARK}}$ and proof Π_{SNARK}. The SNARK proof system (P, V) is defined for the language L where $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi, \mathsf{CRS}_{\mathcal{SS}})$ are instances in the language with witnesses $\Pi_{\mathcal{SS}}$ such that:

$$V_{\mathcal{SS}}(\mathsf{CT}_1, \mathsf{CT}_2, \Pi_{\mathcal{SS}}, \mathsf{CRS}_{\mathcal{SS}}) = 1 \text{ and } h_\Pi = \mathcal{H}(\Pi_{\mathcal{SS}}) \tag{2}$$

2. If the verifier rejects then output 0; otherwise, output $f(\mathsf{Decrypt}_{\mathsf{PKE}}(\mathsf{CT}_1, \mathsf{SK}^1_{\mathsf{PKE}}))$.

Figure 2 The template of a program that is obfuscated during the KeyGen operation.

We now describe the construction of functional encryption for Turing machines.

Construction:

- $\mathsf{Setup}_{\mathsf{FE}}(1^\lambda)$: The $\mathsf{Setup}_{\mathsf{FE}}$ algorithm takes the security parameter λ and computes the following.

1. Generate $(\mathsf{PK}^1_{\mathsf{PKE}}, \mathsf{SK}^1_{\mathsf{PKE}}) \leftarrow \mathsf{Setup}_{\mathsf{PKE}}(1^\lambda)$ and $(\mathsf{PK}^2_{\mathsf{PKE}}, \mathsf{SK}^2_{\mathsf{PKE}}) \leftarrow \mathsf{Setup}_{\mathsf{PKE}}(1^\lambda)$.
2. Set $\mathsf{CRS}_{\mathcal{SS}} \leftarrow \mathsf{Setup}^{\mathcal{SS}}_{\mathsf{NIZK}}(1^\lambda)$ and $\mathsf{CRS}_{\mathsf{SNARK}} \leftarrow \mathsf{Setup}_{\mathsf{SNARK}}(1^\lambda)$.

It sets the public parameters and master secret key as

$$\mathsf{PP} = \{\mathsf{PK}^1_{\mathsf{PKE}}, \mathsf{PK}^2_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}}\} \text{ and } \mathsf{MSK} = \{\mathsf{SK}^1_{\mathsf{PKE}}\}$$

- $\mathsf{KeyGen}_{\mathsf{FE}}(\mathsf{MSK}, f)$: Output a differing-inputs obfuscation $\mathsf{P1}_{\mathsf{obf}}$ corresponding to the program $\mathsf{P1} = \mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ using the size of the Turing machine to be equal to the value $\max\{|\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}|, |\mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}|\}$, where $|\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}|$ (resp., $|\mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}|$) denotes the size of the Turing machine representing $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ (resp., $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$). Also, we adjust the running time of both the programs $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ and $\mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ to be the same [6]. We output the secret key SK_f as the obfuscated Turing machine.

- $\mathsf{Encrypt}_{\mathsf{FE}}(\mathsf{PP}, x \in \{0,1\}^n)$: Execute the following steps to compute the ciphertext corresponding to the encryption of x.
  1. Compute $e_1 = \mathsf{Encrypt}_{\mathsf{PKE}}(\mathsf{PK}^1_{\mathsf{PKE}}, x; r_1)$ and $e_2 = \mathsf{Encrypt}_{\mathsf{PKE}}(\mathsf{PK}^1_{\mathsf{PKE}}, x; r_2)$.
  2. Generate a proof $\Pi_{\mathcal{SS}}$ using the SS-NIZK prover $P^{\mathcal{SS}}_{\mathsf{NIZK}}$ that the encryptions e_1 and e_2 are correctly computed defined for relation $R_{\mathcal{SS}}$.
  3. Compute a hash, using $\mathcal{H}$, of the proof $\Pi_{\mathcal{SS}}$. Denote the hash by $h_{\Pi_{\mathcal{SS}}}$.
  4. Generate a SNARK proof Π_{SNARK} using P_{SNARK} that the proof $\Pi_{\mathcal{SS}}$ is correctly computed. Prover P_{SNARK} takes as input, instance $(e_1, e_2, h_{\Pi_{\mathcal{SS}}})$ along with $\mathsf{CRS}_{\mathcal{SS}}$ and witness $\Pi_{\mathcal{SS}}$.

  Then, the encryption algorithm outputs the ciphertext $(e_1, e_2, c_{\Pi_{\mathcal{SS}}}, \Pi_{\mathsf{SNARK}})$.

- $\mathsf{Decrypt}_{\mathsf{FE}}(\mathsf{SK}_f, c = (e_1, e_2, c_{\Pi_{\mathcal{SS}}}, \Pi_{\mathsf{SNARK}}))$: The decryption algorithm runs the obfuscated program SK_f on input $(e_1, e_2, c_{\Pi_{\mathcal{SS}}}, \Pi_{\mathsf{SNARK}})$ and outputs the answer.

The above functional encryption scheme satisfies the succinctness and the input-specific runtime properties, which is shown in Appendix D. Consider the following lemma. The lemma shows that the class $\mathcal{P}$ is a differing-inputs circuit family. To do this, we first need to define the sampling algorithm $\mathsf{Sampler}_{\mathcal{P}}$ corresponding to $\mathcal{P}$. The sampler on input security parameter, first executes Setup of the SNARK proof system to obtain $\mathsf{CRS}_{\mathsf{SNARK}}$ and then it executes the Setup of the SS-NIZK proof system to obtain $\mathsf{CRS}_{\mathcal{SS}}$. Finally it executes the setup algorithm of the PKE system to obtain $(\mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{PK}^1_{\mathsf{PKE}})$. The sampler outputs the programs $\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{FHE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ and $\mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{FHE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$. We prove the following lemma whose proof can be found in Appendix D.1.

Lemma 2. *Consider the class of programs $\mathcal{P}$ defined as before. Let $\mathsf{P1}_{\mathsf{obf}} = \mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{FHE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ and $\mathsf{P1}_{\mathsf{obf}} = \mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{FHE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ along with the auxilary information aux be the output of the $\mathsf{Sampler}_{\mathcal{P}}$. There does not exist any PPT adversary $\mathcal{A}$ on input $(\mathsf{P1}, \mathsf{P2}, \mathsf{aux})$ outputs y such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$, with non-negligible probability.*

Using this lemma, we now prove the correctness and the security of the FE scheme.

Correctness. We sketch the proof that the above scheme satisfies correctness. We first argue that the program $\mathsf{P1}$ correctly decrypts the FE ciphertext. Then, from the correctness of the

[6] This can be done by making sure that the PKE decryption algorithm as well as the SNARK verifier take worst case running time.

differing-inputs obfuscation it follows that the output of the obfuscated program is a correct decryption of FE ciphertext. The correctness of the SS-NIZK as well as the SNARK implies that the proof, that verifies that the ciphertexts are correctly computed, is accepted by the verifier in $\mathsf{P1}$. Since the proof is accepted by the verifier, the output of the program $\mathsf{M1}$ is the output of the decryption of the PKE ciphertext, which is part of the FE ciphertext. The correctness of the PKE scheme ensures that the PKE ciphertext, and hence the FE ciphertext, is correctly decrypted.

Security proof. We prove the security of the FE construction in this section. We first describe the hybrids and then we argue the computational indistinguishability of the hybrids that will complete the proof. Again, the structure of the hybrids follow the structure of the hybrids in Garg et al. [GGH+13b]. For completeness sake, we present the hybrids here. In the following sequence of hybrids, we move step by step from the indistinguishability game, described in Section A.2, where x_0 is encrypted to the indistinguishability game where x_1 is encrypted. Then, by showing the computational indistinguishability of the hybrids in Appendix D.1 we show that the FE scheme is secure.

Hybrid_5: This hybrid represents the honest execution in the indistinguishability game in Appendix A.2 in which the challenger encrypts the message x_0 in the ciphertext.

Hybrid_6: In this hybrid, the generation of the PKE keys as well as the PKE encryption of the message x_0 is done as in the previous hybrid. The difference between this hybrid and Hybrid_5 is in the generation of the proofs – unlike the previous hybrid, the SS-NIZK proof is simulated here.

More formally, during the setup phase, along with executing other steps honestly, we execute $\mathsf{Sim}_{\mathcal{SS}}$ to obtain "fake" $\mathsf{CRS}_{\mathcal{SS}}$, where $\mathsf{Sim}_{\mathcal{SS}}$ is the simulator of $(P_{\mathcal{SS}}, V_{\mathcal{SS}})$. The keys of the PKE scheme are generated honestly. The adversary on input the public parameters then sends the messages x_0 and x_1 to the challenger. The challenger then encrypts x_0 honestly using the encryption of the PKE scheme, as described the FE scheme, to obtain e_1 and e_2. It then uses the simulator $\mathsf{Sim}_{\mathcal{SS}}$ to produce a simulated proof $\Pi_{\mathcal{SS}}$ for the statement that e_1 and e_2 are correctly computed. The rest of the hybrid is the same as the previous hybrid.

Hybrid_7: In this hybrid, instead of encrypting x_0 both using $\mathsf{PK}^1_{\mathsf{PKE}}$ and $\mathsf{PK}^2_{\mathsf{PKE}}$, we encrypt x_0 using $\mathsf{PK}^1_{\mathsf{PKE}}$ and then encrypt x_1 using $\mathsf{PK}^2_{\mathsf{PKE}}$. The rest of the hybrid, including the simulation of the proofs part, is the same as the previous hybrid.

$\mathsf{Hybrid}_{8,i}$ for $i \in [0, q]$: We now describe a sequence of hybrids, one defined for each query the adversary makes. We move from one hybrid to another by changing the secret key, generated as part of the functional key, used to perform decryption. In $\mathsf{Hybrid}_{8,i}$ the first i queries will result in functional keys generated as obfuscations of the program $\mathsf{P}^{(f_i, \mathsf{SK}^2_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$, where f_i is the i^{th} function queried. The remaining $i+1$ to q queries are generated using the obfuscations of the program $\mathsf{P}^{(f_i, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$ as in hybrid Hybrid_2. Also, the ciphertext in the challenge message is generated as in Hybrid_7. Note that $\mathsf{Hybrid}_{8,0}$ is equivalent to Hybrid_7.

Hybrid_9: This hybrid is the same as $\mathsf{Hybrid}_{8,q}$ with the only difference being that in this case, the challenge ciphertext is generated as encryptions of x_1 under the public keys $\mathsf{PK}^1_{\mathsf{PKE}}$ and $\mathsf{PK}^1_{\mathsf{PKE}}$ respectively. The rest of the hybrid, including the simulation of the NIZK proofs, is the same as $\mathsf{Hybrid}_{8,q}$.

$\mathsf{Hybrid}_{10,i}$ for $i \in [0, q]$: Again, we describe a sequence of hybrids one defined for each query the adversary makes. In $\mathsf{Hybrid}_{10,i}$, the first i private keys requested will result in prvate keys generated as obfuscations of the program $\mathsf{P}^{(f_i, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$. The rest of the private keys, namely from $i+1$ to q, are generated using the program $\mathsf{P}^{(f_i, \mathsf{SK}^2_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$ as in Hybrid_9. Note that $\mathsf{Hybrid}_{10,0}$ is equivalent to Hybrid_9.

Hybrid_{11}: The hybrid is identical to Hybrid_5 with the only difference being in the generation of $\mathsf{CRS}_{\mathcal{SS}}$. In this hybrid, the $\mathsf{CRS}_{\mathcal{SS}}$ is generated from an honest run of the $\mathsf{Setup}_{\mathcal{SS}}$ algorithm and that the SS-NIZK proof $\Pi_{\mathcal{SS}}$ is generated from honestly using the prover $P_{\mathcal{SS}}$. This corresponds to the security game when message x_1 is encrypted for the challenge ciphertext.

5 Delegatable functional encryption scheme

The notion of delagatable functional encryption scheme is introduced in this section. Delegatable functional encryption is a functional encryption scheme having the additional operation of delegation of functional keys. We first give an informal description of the delegate operation. Suppose, Alice has a functional key corresponding to some function. Alice decrypting all the messages all by herself is cumbersome. She wants to delegate some specific decryptions to Bob. One way to do that is Alice hands over her key to Bob. However, Bob can now decrypt messages which he is not supposed to. Instead what Alice can do is compute a new key from the functional key it possesses and she can hand over the key to Bob. The key is designed in such a way that Bob can only decrypt messages he is supposed to and nothing more. This is precisely what delegation deals with. We define the class of functions that can be delegated. Suppose, Alice has a key corresponding to function f then she can delegate those functions g which can be written as a composition of f' on f, denoted by $f' \circ f$, for some function f' [7].

5.1 Definition

We define a delegatable functional encryption scheme to consist of the following PPT algorithms ($\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Delegate}$). The first four PPT algorithms are the same as in the definition of the functional encryption described in Section A.2.

- $\mathsf{Setup}(1^\lambda)$ - a polynomial time algorithm that takes the unitary representation of the security paramter λ and outputs a public paramteres PP and a master secret key MSK.
- $\mathsf{KeyGen}(\mathsf{MSK}, f)$ - a polynomial time algorithm that takes as input the master secret key MSK and a function f implementable by a Turing machine $M \in \mathcal{M}$ and outputs a corresponding secret key SK_f.
- $\mathsf{Encrypt}(\mathsf{PP}, x)$ - a polynomial time algorithm that takes the public parameters PP and a string $x \in \mathcal{S}$ and outputs a ciphertext CT.
- $\mathsf{Decrypt}(\mathsf{SK}_f, \mathsf{CT})$ - a polynomial time algorithm that takes a secret key SK_f and ciphertext encrypting message $x \in \mathcal{S}$ and outputs $f(x)$.
- $\mathsf{Delegate}(\mathsf{PP}, \mathsf{SK}_f, f')$ - a polynomial time algorithm that takes as input a public key PP, a functional key SK_f and a function f' and outputs a functional key $\mathsf{SK}_{f' \circ f}$ that evaluates the function $f' \circ f$ on the message contained in the ciphertext.

A delegatable functional encryption scheme satisfies two main properties, namely correctness and security. The criterion for correctness is the same as that of the functional encryption scheme. In addition, the following must be satisfied – if the output of a delegate operation on input SK_f and f' is $\mathsf{SK}_{f' \circ f}$ then the decryption algorithm on input $\mathsf{SK}_{f' \circ f}$ along with an encryption of a message x should give $f'(f(x))$ as its output. We describe the security notion next.

[7] More formally, $f' \circ f$ takes as input x and outputs $f'(f(x))$.

5.2 Security notion

We now describe the security notion employed for a delegatable encryption scheme. We follow the security notion defined in [SW08] for a predicate encryption scheme. The security is modelled as a game between a challenger and an adversary.

Setup. The challenger executes the Setup algorithm of the delegatable functional encryption scheme and gives the public key, denoted by PK to the adversary.
Query. The adversary submits queries to the challenger adaptively. There are three subphases in the query phase. The first is the creation of the functional key, second is the delegation phase and the third is the reveal phase.

- *Creation.* The adversary submits the queries f_i to the challenger who computes the keys SK_{f_i} corresponding to f_i. These keys are not yet revealed to the adversary.
- *Delegation.* The adversary now chooses the keys, generated during the creation phase, on which the delegation operation need to be applied. This is done by the adversary submitting a function f_i' along with an index i, to indicate the key on which the delegate operation need to applied. The challenger then executes $\mathsf{Delegate}$ on SK_{f_i} along with f_i' to obtain $\mathsf{SK}_{f_i' \circ f_i}$. As in the previous case, the key is not yet revealed to the adversary.
- *Reveal.* The adversary asks the challenger to reveal a functional key that was generated in one of the previous phases.

Challenge. The adversary then sends the two challenge messages x_0, x_1 such that for all functional keys SK_f created by the challenger (including the ones during the delegation phase), $f(x_0) = f(x_1)$. If this condition is not satisfied then the challenger aborts the game. Otherwise, the challenger encrypts x_b using the public key PK, where b is a bit chosen uniformly at random, and the resulting ciphertext is then handed over to the adversary.
Query. This phase is similar to the previous query phase. In this phase too, for any functional key SK_f created, $f(x_0)$ should be the same as $f(x_1)$.

Guess. The game ends when the adversary guesses a bit b'. The advantage of an adversary in the above game is defined to be $|\mathsf{Prob}[b' = b] - \frac{1}{2}|$.

Definition 5. *A delegatable functional encryption scheme is said to be (fully) secure if for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ is a negligible function of λ.*

5.3 Construction

Consider the functional encryption scheme described in Section 4. Corresponding to this scheme we define a delegate operation, denoted by $\mathsf{Delegate}$, as follows. On input a key SK_f and a function f', the delegate operation computes the differing-inputs obfuscation of the program $\mathsf{P}^{f',\mathsf{SK}_f}$. The program $\mathsf{P}^{f',\mathsf{SK}_f}$ on input $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi, \varphi)$, first evaluates the obfuscation SK_f on input $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi)$ to obtain z. It then evaluates f' on z to obtain $f'(z)$, which it then outputs.

We claim that this is a delegatable encryption scheme. The correctness of this scheme follows from the correctness of differing-inputs obfuscation. We argue about the security informally here. We first design the hybrids such that the first hybrid corresponds to the indistinguishability game in the delegatable functional encryption scheme and the last hybrid corresponds to the game in the functional encryption scheme. In each hybrid, we replace a delegate operation by a key generation operation. That is, if an adversary requests delegate operation f' on key SK_f, instead of performing the delegation operation we generate a fresh key $\mathsf{SK}_{f' \circ f}$.

To argue the indistinguishability of the hybrids, note that it suffices to show that the output distribution of the key generation for the function $f' \circ f$ is computationally indistinguishable from the output distribution of the delegation operation on input SK_f, corresponding to f, and f'. To see this, observe that the programs $\mathsf{P}^{f',\mathsf{SK}_f}$ and $\mathsf{P}^{(f,\mathsf{SK}_1,\mathsf{CRS}_{SS},\mathsf{CRS}_{\mathsf{SNARK}})}$ are equivalent. The output of the key generation is the obfuscation of $\mathsf{P}^{(f,\mathsf{SK}_1,\mathsf{CRS}_{SS},\mathsf{CRS}_{\mathsf{SNARK}})}$ and correspondingly the output of the delegate operation is the output of $\mathsf{P}^{f',\mathsf{SK}_f}$. From the equivalence of these programs, it follows that their obfuscations, and hence the outputs (distributions) of KeyGen and Delegate are computationally indistinguishable. This proves that any two consecutive hybrids are computationally indistinguishable. The adversary can succeed in the last hybrid with only negligible probability and this follows from the fact that our functional encryption scheme is secure. Hence, the adversary can succeed in the first hybrid, which is the security game of the delegatable encryption scheme, only with negligible probability. This completes the proof.

Remark. The above delegatable functional encryption scheme works for both circuits as well as Turing machines. If a delegatable scheme need to be constructed only for circuits then we can directly construct the scheme from the functional encryption scheme by Garg et. al. [GGH+13b] using the delegation operation defined as above. More specifically, instead of using differing-inputs obfuscation we can directly use indistinguishable obfuscation for circuits.

References

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*, pages 111–120. ACM, 2013.

[BCP13] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. Cryptology ePrint Archive, Report 2013/650, 2013. `http://eprint.iacr.org/`.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. pages 1–18, 2001.

[BGK+13] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. Cryptology ePrint Archive, Report 2013/631, 2013. `http://eprint.iacr.org/`.

[BR13] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. Cryptology ePrint Archive, Report 2013/563, 2013. `http://eprint.iacr.org/`.

[BZ13] Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. Cryptology ePrint Archive, Report 2013/642, 2013. `http://eprint.iacr.org/`.

[Dam90] Ivan Bjerre Damgård. A design principle for hash functions. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 416–427. Springer, 1990.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *EUROCRYPT*, 2013.

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *Proc. of FOCS (to appear)*, 2013.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *STOC*, 2013.

[GK03] Shafi Goldwasser and Yael Tauman Kalai. On the (in) security of the fiat-shamir paradigm. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 102–113. IEEE, 2003.

[GKP+13] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, , and Nickolai Zeldovich. How to run turing machines on encrypted data. In *CRYPTO*, 2013.

[Mer88] Ralph Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1988.

[Mer90] Ralph C Merkle. One way hash functions and des. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 428–446. Springer, 1990.

[PF79] Nicholas Pippenger and Michael J Fischer. Relations among complexity measures. *Journal of the ACM (JACM)*, 26(2):361–381, 1979.

[SW08] Elaine Shi and Brent Waters. Delegating capabilities in predicate encryption systems. In *Automata, Languages and Programming*, pages 560–578. Springer, 2008.

A Indistinguishability Obfuscation for Turing machines and FE for Turing machines: Definitions

A.1 $\mathsf{i}\mathcal{O}$ for Turing machines

We define the notion of indistinguishability obfuscation for Turing machines similar to the definition of indistinguishability obfuscation for circuits. Note that the two Turing machines that need to be obfuscated in the security game, should not only be identical on all inputs but they also need to having the same running time on all the inputs. Note that the definition of differing-inputs obfuscation for Turing machines implies the following definition and hence our construction in Section 3 also satisfies this definition.

Definition 6. ***(Indistinguishable Obfuscators for Turing machines)*** *A uniform PPT machine $\mathsf{i}\mathcal{O}_{\mathsf{TM}}$ is called an Indistinguishable Obfuscators for the Turing machine family $\mathcal{M}$, if the following conditions are satisfied:*

- ***Correctness****: For all security parameters $\lambda \in \mathbb{N}$, for all $M \in \mathcal{M}$, for all inputs x, we have that*
$$\mathsf{Prob}[M'(x) = M(x) \ : \ M' \leftarrow \mathsf{i}\mathcal{O}(\lambda, M)] \ = \ 1$$
- ***Indistinguishable Obfuscation****: For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all $M_0, M_1 \in \mathcal{M}, \mathsf{aux}$ such that for all x, $M_0(x) = M_1(x)$ and $\mathsf{time}(M_0, x) = \mathsf{time}(M_1, x)$ we have that*
$$|\mathsf{Prob}[D(\mathsf{i}\mathcal{O}(\lambda, M_0, \mathsf{aux})) = 1] \ - \ \mathsf{Prob}[D(\mathsf{i}\mathcal{O}(\lambda, M_1, \mathsf{aux})) = 1] \leq \alpha(\lambda)$$

In addition to the above properties if $\mathsf{i}\mathcal{O}_{\mathsf{TM}}$ satisfies the following properties, with respect to a universal polynomial p, then we say that $\mathsf{i}\mathcal{O}_{\mathsf{TM}}$ is ***succinct*** *and has* ***input-specific run time****.*

- ***Succinct****: The size of M' is $p(\lambda, |M|)$, where $|M|$ denotes the size of the Turing machine M.*
- ***Input-specific run time****: The running time of M' on an input x is $p(\lambda, \mathsf{time}(M, x))$.*

A.2 Functional Encryption for Turing machines

We cast the definition of functional encryption in Garg et al. [GGH+13b] for the case of Turing machines. Though the functional encryption for Turing machines have already been defined by Goldwasser et al. [GKP+13], our definition differs from their definition in many ways – (i) single key versus many key queries, (ii) simulation based versus indistinguishability game based and so on. While defining FE for Turing machines we restrict the adversary to make only certain type of function queries which is based on the running time of the function. This is to avoid trivial attacks where the attacker will be able to distinguish the encryptions of two messages by choosing a function query whose running time is significantly different on both the messages.

Definition 7. *Let the message space be $\mathcal{S} = \mathcal{S}_\lambda$. A functional encryption scheme defined for a family of Turing machines $\mathcal{M}_\mathcal{T}$, parameterized by a Turing machine $\mathcal{T}$, consists of four algorithms* $\mathsf{FE} = \{\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}\}$*:*

- $\mathsf{Setup}(1^\lambda)$ - *a polynomial time algorithm that takes the unitary representation of the security paramter λ and outputs a public paramteres* PP *and a master secret key* MSK*.*
- $\mathsf{KeyGen}(\mathsf{MSK}, f)$ - *a polynomial time algorithm that takes as input the master secret key* MSK *and a function f implementable by a Turing machine $M \in \mathcal{M}$ and outputs a corresponding secret key* SK_f*.*
- $\mathsf{Encrypt}(\mathsf{PP}, x)$ - *a polynomial time algorithm that takes the public parameters* PP *and a string $x \in \mathcal{S}$ and outputs a ciphertext* CT*.*
- $\mathsf{Decrypt}(\mathsf{SK}_f, \mathsf{CT})$ - *a polynomial time algorithm that takes a secret key* SK_f *and ciphertext encrypting message $x \in \mathcal{S}$ and outputs $f(x)$.*

A functional encryption scheme is correct for $\mathcal{M}$ if for all $M \in \mathcal{M}$ and all messages $x \in \mathcal{S}$:

$$\mathsf{Prob}[(\mathsf{PK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda); \mathsf{Decrypt}(\mathsf{KeyGen}(\mathsf{MSK}, f), \mathsf{Encrypt}(\mathsf{PK}, x)) = f(x)] = \mathsf{negl}(\lambda)$$

We now define the (fully) indistinguishability security for functional encryption which is described in form of an indistinguishability game between an attacker $\mathcal{A}$, whose running time is upper bounded by λ^c for a constant c, and a challenger to whom the constant c is given.

Setup: The challenger runs $(\mathsf{PK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda)$ and gives PP to $\mathcal{A}$.

Query: A submits queries $f_i \in \mathcal{M}$. We assume, without loss of generality that f_i can be represented by a Turing machine M_i which, on an input x, outputs $f_i(x)$ along with the taken by M_i to execute on x. The adversary A is then given $\mathsf{SK} \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, f_i)$

Challenge: The adversary on input a security parameter outputs messages (x_0, x_1).

Query: A executes another query phase. It submits queries of the form $f_i \in \mathcal{M}$ which are represented by Turing machines M_i as in the previous Query phase. If $f_i(x_0) = f_i(x_1)$, adversary A is given $\mathsf{SK} \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, f_i)$ else the game is aborted. As in the previous definitions, we assume that the description of M_i contains a time bound τ such that $M_i(x) \leq \tau$ for all inputs x.

Guess: A eventually outputs a bit b' in $\{0, 1\}$.

The advantage of an adversary $\mathcal{A}$ is defined to be $|\mathsf{Prob}[b' = b] - \frac{1}{2}|$.

Definition 8. *A functional encryption scheme is (fully) indistinguishability secure if for any PPT adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ in the above indistinguishability game is negligible.*

In addition to the above properties, we also consider the following two properties for functional encryption schemes for Turing machines.

- **Succinctness**: A functional encryption scheme is said to be succinct if the functional key generated using KeyGen for the function f is $p(\lambda, |M|)$, where p is a polynomial and M denotes the size of the Turing machine representing the function f.
- **Input-specific run time**: A functional encryption scheme is said to have input-specific run time if the decryption algorithm on input a functional key for a function f along with an encryption of x, takes time $p(\lambda, \mathsf{time}(M, x))$, where M is the Turing machine representing the function f.

B Background

B.1 Fully Homomorphic Encryption

We define the notion of fully homomorphic encryption (FHE) scheme. It consists of four PPT algorithms ($\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Eval}$) defined as follows.

- $\mathsf{KeyGen}(1^\lambda)$: On input a security parameter 1^λ it outputs a public key $\mathsf{PK_{FHE}}$ and a decryption key $\mathsf{SK_{FHE}}$.
- $\mathsf{Encrypt}(m, \mathsf{PK_{FHE}})$: On input a message m and public key $\mathsf{PK_{FHE}}$ it outputs a ciphertext denoted by CT.
- $\mathsf{Decrypt}(\mathsf{CT}, \mathsf{SK_{FHE}})$: On input a ciphertext and a decryption key $\mathsf{SK_{FHE}}$ it outputs a message m.
- $\mathsf{Eval}(\mathsf{CT}, \mathsf{PK_{FHE}}, f)$: On input a ciphertext, a public key [8] and a function f, outputs another ciphertext CT' such that the decryption of CT' yields the message $f(m)$.

The security of FHE is defined very similar to the security of the IND-CPA public key encryption scheme. There does not exist any PPT adversary A such that, for any pair of messages m_0, m_1, the probability that on input $\mathsf{Encrypt}(m_0, \mathsf{PK_{FHE}})$ it outputs 0 (resp., 1) is negligibly close to the probability that on input $\mathsf{Encrypt}(m_1, \mathsf{PK_{FHE}})$ it outputs 1.

B.2 FHE for Turing machines

We present the construction of the compiler verbatim from Goldwasser et. al. [GKP$^+$13] below. The compiler, denoted by $\mathsf{Compile_{FHE}}$, takes as input a Turing machine M and a number of steps t, and produces a Turing machine that computes the FHE evaluation of M for t steps. Let $\widehat{x}$ denote the FHE encryption of x.

$\mathsf{Compile_{FHE}}(M, t)$:

- First, transform M into an oblivious Turing machine M_O by applying the Pippenger-Fischer transformation [PF79] for time bound t. This transformation results in a new Turing machine M_O and a transition function δ for M_O. Namely, δ takes as input tape input bit b, a state state and outputs a new state state', new content b' for the tape location, and bit next indicating whether to move left or right; namely $\delta(b, \mathsf{state}) = (\mathsf{state}', b', \mathsf{next})$. Let the movement function next be such that $\mathsf{next}(i)$ indicates whether the head on the input tape of M_O should move left or right after step i.
- Based on (M_O, next), construct a new Turing machine M_{FHE} that takes as input a FHE public key $\mathsf{PK_{FHE}}$ and an input encryption $\widehat{x}$. M_{FHE} applies the transition function δ_{FHE} (the FHE evaluation of δ using $\mathsf{PK_{FHE}}$) t times. Each cell of the tapes of M_O corresponds to an FHE encrypted value for M_{FHE}. The state of M_{FHE} at time i is the FHE encryption

[8]It is not always necessary that we need to use the public key for FHE evaluation. Sometimes, a separate key for FHE evaluation alone is also used.

of the state of M_O corresponds to an FHE encrypted value for M_{FHE}. The state of M_{FHE} at time i is the FHE encryption of the state of M_O at time i. At step i, the transition function δ_{FHE} takes as input the encrypted bit from the input tape $\widehat{b}$ that the head currently points at, the current encrypted state $\widehat{\mathsf{state}}$ and outputs an encrypted new state $\widehat{\mathsf{state}'}$ and a new content $\widehat{b'}$. To determine whether to move the head left of right, compute $\mathsf{next}(i)$.

- Output the description of M_{FHE}.

The running time of $\mathsf{Compile}_{\mathsf{FHE}}$ and M_{FHE} is polynomial in t. The Turing machine M_{FHE} takes as input a public key and a ciphertext and then performs the FHE evaluation of M on the ciphertext. The resulting answer is then output by M_{FHE}.

B.3 Succinct Non Interactive Arguments of Knowledge

We now give background for succinct non-interactive arguments of knowledge. We present the details verbatim from Goldwasser et al. [GKP$^+$13]. We define the universal relation as a canonical form to represent verification-of-computation problems.

Definition 9. *[BCCT13] The universal relation is the set R_U of instance-witness pairs $(y, w) = ((U, x, t), w)$, where $|y|, |w| \leq t$ and U is a Turing machine, such that U accepts (x, w) after at most t steps. We denote by L_U the universal language corresponding to R_U. For any $c \in \mathbb{N}$, the universal NP relation is the set $R_{U,c}$, defined as R_U with the additional constraint that $t \leq |x|^c$.*

A SNARK is a triple of algorithms (Setup, P, V) that works as follows.

- The generator Setup on input the security parameter λ, samples a reference string CRS (since we consider publicly verifiable SNARKs, the CRS can also contain the public verification state). The Setup also takes as input a time bound B but we set this to $B = \lambda^{\mathsf{log}\lambda}$ which will never be achieved for NP language. Therefore, for simplicity, we do not make B explicit from now on.
- The honest prover $P(\mathsf{CRS}, y, w)$ produces a proof π for the statement $y = (U, x, t)$ given a valid witness w.
- The verifier $V(\mathsf{CRS}, y, \pi)$ takes as input the CRS, the instance y and a proof π and deterministically verifies π.

The SNARK is adaptive if the prover may choose the statement after seeing CRS.

Definition 10. *A triple of algorithms (Setup, P, V) for the relation $R_{U,c}$, where Setup is probabilistic and V is deterministic, is a SNARK if the following conditions are satisfied:*

- ***Completeness**: For every large enough security parameter $\lambda \in \mathbb{N}$, and for every instance-witness pair $(y, w) = ((U, x, t), w) \in R_U$,*

$$\mathsf{Prob}[\mathsf{CRS} \leftarrow \mathsf{Setup}(1^\lambda);\ \pi \leftarrow P(\mathsf{CRS}, y, w)\ :\ V(\mathsf{CRS}, y, \pi)) = 1]$$

- ***Proof of knowledge**: For every polynomial-size prover P^* there exists a polynomial-size extractor Ext such that for every large enough security parameter $\lambda \in \mathbb{N}$, every auxillary input $z \in \{0, 1\}^{poly(\lambda)}$, and every constant c:*

$$\mathsf{Prob}[\mathsf{CRS} \leftarrow \mathsf{Setup}(1^\lambda); (y, \pi) \leftarrow P^*(\mathsf{CRS}, z);\ w \leftarrow \mathsf{Ext}(\mathsf{CRS}, z)\ :$$
$$V(\mathsf{CRS}, y, \pi) = 1 \text{ and } (y, w) \notin R_{U,c}]\ =\ negl(\lambda).$$

- ***Efficiency**. There exists a universal polynomial p such that, for every large security parameter λ and every instance $y = (M, x, t)$,*
  1. *The generator $\mathsf{Setup}(1^\lambda)$ runs in time $p(\lambda)$;*

2. The prover $P(\mathsf{CRS}, y, w)$ runs in time $p(\lambda, |U|, |x|, t)$;

3. The verifier $V(\mathsf{CRS}, y, \pi)$ runs in time $p(\lambda, |U|, |x|)$;

4. An honestly generated proof has size $p(\lambda)$.

Bitansky et al. [BCCT13] demonstrated a SNARK proof system under knowledge of exponent assumptions.

B.4 IND-CPA secure PKE

An IND-CPA (or semantically) secure Public Key Encryption scheme consists of three PPT algorithms ($\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}$) described as follows.

1. $\mathsf{KeyGen}(1^\lambda)$: On input 1^λ, it outputs public key $\mathsf{PK}_{\mathsf{PKE}}$ and decryption key $\mathsf{SK}_{\mathsf{PKE}}$.
2. $\mathsf{Encrypt}(m, \mathsf{PK}_{\mathsf{PKE}})$: On input message m and the public key, it outputs a ciphertext CT.
3. $\mathsf{Decrypt}(\mathsf{CT}, \mathsf{SK}_{\mathsf{PKE}})$: On input a ciphertext CT and the decryption key, it outputs m.

The IND-CPA scheme is said to be semantically secure if for any PPT adversary $\mathcal{A}$, there exists a negligible function α such that the following is satisfied for any two messages m_0, m_1 and for $b \in \{0, 1\}$:

$$|\mathsf{Prob}[\mathcal{A}(1^\lambda, \mathsf{Encrypt}(m_0, \mathsf{PK}_{\mathsf{PKE}})) = b] - \mathsf{Prob}[\mathcal{A}(1^\lambda, \mathsf{Encrypt}(m_1, \mathsf{PK}_{\mathsf{PKE}})) = b]| \leq \alpha(\lambda)$$

B.5 Simulation sound NIZK

We define the notion of simulation sound non-interactive zero knowledge, which is a specific type of NIZK proof system. Intuitively, this notion says that there does not exist any efficient adversary even after receiving "fake" proofs for statements of his choice he cannot output any convincing proof, including fake proofs, for a statement for which he had not received any proof before.

More formally, a simulation-sound NIZK satisfies the following property along from the completeness, soundness and zero knowledge properties of any NIZK proof system. Consider the following game defined for PPT any adversary $\mathcal{A}$. The game begins with the execution of the simulator of the NIZK proof system who generates a fake CRS and a corresponding trapdoor. Then, $\mathcal{A}$ is given oracle access to a simulator which has a corresponding trapdoor. The adversary can submit any statement to this oracle and he will correspondingly get back a convincing proof (which is accepted by the NIZK verifier). The game ends with $\mathcal{A}$ outputting (x, Π). The adversary wins the game if (1) x was not equal to any of the statements he had queried the oracle and (2) x is not in the language for which the NIZK is defined and (3) Π is accepted by the NIZK verifier. We now define the simulation soundness property of a NIZK proof system.

Definition 11. *A NIZK proof system is said to satisfy simulation soundness if $\mathcal{A}$ wins the above game with negligible probability.*

C Succinctness and Input-specific time of $\mathsf{di}\mathcal{O}$ for TMs

Size of the Obfuscation: We now upper bound the size of the obfuscated Turing machine which is obtained by inputting the Turing machine M to the $\mathsf{Obfuscate}_{\mathsf{TM}}$ algorithm. Denote the output of the $\mathsf{Obfuscate}_{\mathsf{TM}}$ algorithm to be $(\mathcal{P}1_{\mathsf{obf}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}, g_1, g_2, \mathsf{CRS})$.

- The size of the FHE public keys $(\mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}})$ can be upper bounded by a polynomial in the security parameter.

- The size of the FHE encryptions g_1 and g_2 depends on the size of the message, which is in this case, Turing machine M along with the security parameter. That is, the size of g_1 and g_2 can be upper bounded by a polynomial in $(\lambda, |M|)$.
- The size of the common reference string CRS is a polynomial in the security parameter (refer to Appendix B.3).
- The size of the obfuscation $\mathcal{P}1_{\mathsf{obf}}$ is a polynomial in $(\lambda, |\mathsf{P1}|)$, where $|\mathsf{P1}|$ represents the size of the circuit of $\mathsf{P1}$. Program $\mathsf{P1}$ consists of two main components – the SNARK verifier circuit and the decryption circuit. The size of the SNARK verifier circuit is a polynomial in its inputs. The inputs to the SNARK verifier circuit consists of the following.
  - A pair of encryptions of Turing machine M. The size of this is a polynomial in $(\lambda, |M|)$.
  - Encryptions of a single bit of the output of the Turing machine. The size of this is just a polynomial in λ.
  - Index of the output bit. The size of this is logarithmic in the running time of the Turing machine. Since we are only interested with efficient Turing machines, we can loosely upper bound this quantity by the security parameter λ.
  - Iteration number t. Observe that this too can be can be loosely upper bounded by the security parameter λ.

  The decryption circuit on the other hand takes a ciphertext corresponding to the encryption of a single bit and hence, the size of the decryption circuit is a polynomial in the security parameter. Combining all the facts, we get $|\mathsf{P1}|$ to be a polynomial in $(\lambda, |M|)$.

Running time of the Evaluate algorithm: We first make the following observation. Suppose the Turing machine M takes time T to execute on input x then the number of iterations in the evaluation procedure need to be performed for $O(2T\mathsf{log}T)$ times. We then determine the amount of time taken in each iteration t step by step.

- In the first step, the FHE compiler is executed which takes time which is a polynomial in 2^t, and hence at most time T, and the size of the universal Turing machine $\mathsf{UTM}^i_{(x,2^t)}$ (Refer Appendix B.2). Further, the size of the universal Turing machine is a polynomial in $|x|$ and the security parameter λ. Hence, the running time of the compiler is essentially a polynomial in $(\lambda, T, |x|)$.
- In the second step, the hash of the input x is computed. The running time of this step is a polynomial in $(|x|, \lambda)$.
- In the third step, a SNARK proof is computed and the running time of this is nothing but the running time of the SNARK prover. The running time of the SNARK prover is a polynomial in $(\lambda, |M|, |x|, \mathsf{time}(M, x))$.
- In the last step, the obfuscation algorithm is evaluated. Since this obfuscation is in the form of a circuit, the running time of this depends on the size of the obfuscation, which is nothing but a polynomial in $(\lambda, |M|)$.

From the above points, we have that the running time of the Evaluate algorithm is a polynomial in $(\lambda, |M|, x, T)$.

C.1 Proof of differing-inputs of Lemma 1

To prove this, we assume that there exists an adversary $\mathcal{A}$ that outputs y such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$. Using $\mathcal{A}$, we construct another adversary $\mathcal{A}_{\mathcal{M}}$ which violates the differing-inputs property of $\mathcal{M}$ arriving at a contradiction. Before we proceed further, we make a notational simplification. We assume that the input to $\mathsf{P1}$ (or $\mathsf{P2}$) can be parsed as (z, φ), where $z = (i, e_1^{(i)}, e_2^{(i)}, g_1, g_2, h_x, t, \varphi)$.

We present the following claims that will be useful when we calculate the probability of success of $\mathcal{A}_{\mathcal{M}}$. The first claim says that if the programs $\mathsf{P1}$ and $\mathsf{P2}$ differ on any input $y = (z, \varphi)$ then the verifiers both in $\mathsf{P1}$ and $\mathsf{P2}$ accept the proof φ. Recall that any program in $\mathcal{P}$ has two components, namely the low depth SNARK verifier along with FHE decryption circuit. The second claim states that if both the programs differ on any input (z, φ) then the output of $\mathsf{P1}$ (resp., $\mathsf{P2}$) is the decryption of $e_1^{(i)}$ (resp., $e_2^{(i)}$). Hence, as a consequence, we have that the decryption of $e_1^{(i)}$ (with respect to public key $\mathsf{PK}^1_{\mathsf{FHE}}$) different from the decryption of $e_2^{(i)}$ with respect to $\mathsf{PK}^2_{\mathsf{FHE}}$. We now state the claims.

Claim 11. If there exists a $y = (z, \varphi)$ such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$ then the verifier in both $\mathsf{P1}$ and $\mathsf{P2}$ accept φ.
Proof. The first observation is that the verifier as part of $\mathsf{P1}$ is same as the verifer which is part of $\mathsf{P2}$. The second observation is that the verifier in $\mathsf{P1}$ (resp., $\mathsf{P2}$) does not reject the proof φ. This is because, if the verifier in $\mathsf{P1}$ (resp., $\mathsf{P2}$) rejects then the output of $\mathsf{P1}$ (resp., $\mathsf{P2}$) is 0 which will contradict our hypothesis that the output of the two programs are different.

Claim 12. If there exists $y = (i, e_1^{(i)}, e_2^{(i)}, h_x, t, \varphi)$ such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$ then the output of $\mathsf{P1}$ (resp., $\mathsf{P2}$) is the decryption of $e_1^{(i)}$ (resp., $e_2^{(i)}$) with respect to $\mathsf{PK}^1_{\mathsf{FHE}}$ (resp., $\mathsf{PK}^2_{\mathsf{FHE}}$).
Proof. From Claim 11, we have that the verifiers as part of both $\mathsf{P1}$ and $\mathsf{P2}$ accept and hence, the output of $\mathsf{P1}(y)$ is the decryption of $e_1^{(i)}$ with respect to $\mathsf{PK}^1_{\mathsf{FHE}}$ and similarly, the output of $\mathsf{P2}$ is the decryption of $e_2^{(i)}$ with respect to $\mathsf{PK}^2_{\mathsf{FHE}}$.

Now, consider the following adversary. Since (P, V) which is a SNARK system has knowledge extractability property we assume that there exists an extractor $\mathsf{Ext} = (\mathsf{Ext}_1, \mathsf{Ext}_2)$ such that Ext_1 generates $(\mathsf{CRS}, \mathsf{state})$ and then Ext_2 on input an instance, $(\mathsf{CRS}, \mathsf{state})$ along with a proof, extracts a witness corresponding to that instance.

```
$\underline{\mathcal{A}_{\mathcal{M}}(M_0, M_1, \mathsf{aux}_{\mathcal{M}})}$:
$(\mathsf{SK}^1_{\mathsf{FHE}}, \mathsf{PK}^1_{\mathsf{FHE}}) \leftarrow \mathsf{Setup}_{\mathsf{FHE}}(1^\lambda)$
$(\mathsf{SK}^2_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}) \leftarrow \mathsf{Setup}_{\mathsf{FHE}}(1^\lambda)$
$(\mathsf{CRS}, \mathsf{state}) \leftarrow \mathsf{Ext}_1(1^\lambda)$
$g_1 \leftarrow \mathsf{Encrypt}_{\mathsf{FHE}}(\mathsf{PK}^1_{\mathsf{FHE}}, M_0)$
$g_2 \leftarrow \mathsf{Encrypt}_{\mathsf{FHE}}(\mathsf{PK}^2_{\mathsf{FHE}}, M_1)$
Denote $\mathsf{di}\mathcal{O}(\lambda, \mathsf{P}^{(g_1, g_2, \mathsf{CRS})}_{(\mathsf{SK}^1_{\mathsf{FHE}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}})}$ by $\mathsf{P1}$
Denote $\mathsf{di}\mathcal{O}(\lambda, \mathsf{P}^{(g_1, g_2, \mathsf{CRS})}_{(\mathsf{SK}^2_{\mathsf{FHE}}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}})}$ by $\mathsf{P2}$
$y \leftarrow \mathcal{A}(\mathsf{P1}, \mathsf{P2}, \mathsf{PK}^1_{\mathsf{FHE}}, \mathsf{PK}^2_{\mathsf{FHE}}, g_1, g_2, \mathsf{CRS})$
Parse $y$ as $(z, \varphi)$
$x \leftarrow \mathsf{Ext}_2(z, \varphi, \mathsf{CRS}, \mathsf{state})$
Output $x$
```

The next claim shows that if $\mathcal{A}$ can produce an input y such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$ with non-negligible probability then $\mathcal{A}_{\mathcal{M}}$ violates the security game of the differing-inputs corresponding to the family $\mathcal{M}$ (Definition 4). Before we go ahead and prove the claim, we first observe that the distribution of $(\mathsf{P1}, \mathsf{P2}, \mathsf{aux}_{\mathcal{M}})$ as input to $\mathcal{A}$ in the description of $\mathcal{A}_{\mathcal{M}}$ is the same as the output distribution of $\mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}$.

Claim 13. Let $(\mathsf{P1}, \mathsf{P2}, \mathsf{aux}) \leftarrow \mathsf{Sampler}^{\mathcal{M}}_{\mathcal{P}}(1^\lambda)$. If there exists an adversary $\mathcal{A}$ on input

$(\mathsf{P1}, \mathsf{P2}, \mathsf{aux})$ outputs y with non-negligible probability then the adversary $\mathcal{A}_{\mathcal{M}}$ on input (M_0, M_1), which is the output of $\mathsf{Sampler}_{\mathcal{M}}(1^\lambda)$, produces x such that $M_0(x) \neq M_1(x)$.
Proof. Suppose the adversary $\mathcal{A}$ outputs y such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$ with non-negligible probability. We make the following two observations that will prove the above claim.

- Using Claim 11, we have the fact that the verifier accepts the proof φ corresponding to the instance z with non-negligible probability, where y can be parsed as (z, φ) is the output of $\mathcal{A}$. From the knowledge extractability property, we have the fact that the extractor outputs a valid witness x with non-negligible probability. Since, x is a valid witness to z, we have the fact that $e_1^{(i)}$ is an encryption of $M_0(x)$ with respect to public key $\mathsf{PK}^1_{\mathsf{FHE}}$ and similarly, $e_2^{(i)}$ is an encryption of $M_1(x)$ with respect to public key $\mathsf{PK}^2_{\mathsf{FHE}}$.
- Using Claim 12, we have that the output of program $\mathsf{P1}$ (resp., $\mathsf{P2}$) is the decryption of g_1 (resp., g_2) with respect to $\mathsf{PK}^1_{\mathsf{FHE}}$ (resp., $\mathsf{PK}^2_{\mathsf{FHE}}$). Rephrasing this in terms of first observation, the output of $\mathsf{P1}$ on input y is $M_0(x)$ and the output of $\mathsf{P2}$ on input y is $M_1(x)$. Since, y is such that $\mathsf{P1}(y) \neq \mathsf{P2}(y)$ with non-negligible probability, we have that $M_0(x) \neq M_1(x)$. This completes the proof.

The above claim contradicts the assumption that $\mathcal{M}$ is a differing-inputs Turing machine family and this proves $\mathcal{P}$ is a differing-inputs circuit family.

Corollary 1. *Differing-inputs obfuscation for the circuit family $\mathcal{P}$ exists under the assumption that IND-CPA FHE exists, SNARKs exists, collision resistant hash functions and differing-inputs obfuscation exists for any differing-inputs circuit family.*

C.2 Indistinguishability of hybrids Hybrid_0 to Hybrid_4

We present a series of claims that show that the hybrids are computationally indistinguishable with respect to each other.

Claim 1. Hybrids H_0 and H_1 are computationally indistinguishable under the assumption that the FHE scheme is IND-CPA secure.
Proof. We assume that these two hybrids are distinguishable and then arrive at a contradiction by contradicting the IND-CPA security of the FHE scheme. Suppose there exists an adversary $\mathcal{A}$ that distinguishes hybrids Hybrid_0 and Hybrid_1 then we construct an adversary $\mathcal{A}'$ that breaks the IND-CPA security of FHE scheme as follows. The adversary $\mathcal{A}'$, on input a public key PK_1, first executes $\mathcal{A}$ to get the messages M_0 and M_1. It then sends this to the challenger who decides to encrypt either M_0 or M_1 depending on the challenge bit. The challenege ciphertext, $\mathsf{CT}^{(2)}$, is handed over to $\mathcal{A}'$ who does the following. It generates public key-secret key pair $(\mathsf{SK}_0, \mathsf{PK}_0)$ and then encrypts m_0 using PK_0 to obtain $\mathsf{CT}^{(1)}$. Further, it generates the program $\mathsf{P1}$ in which the decryption is done using the decryption key SK_0. It finally gives $(\mathsf{CT}^{(1)}, \mathsf{CT}^{(2)}, \mathcal{P}1_{\mathsf{obf}})$, where $\mathsf{P1}_{\mathsf{obf}} = \mathsf{di}\mathcal{O}(\mathsf{P1})$, to $\mathcal{A}$ and then $\mathcal{A}'$ outputs whatever $\mathcal{A}$ outputs.

Claim 2. Hybrids H_1 and H_2 are computationally indistinguishable under the assumption that differing-inputs obfuscators exist for all circuits.
Proof. Consider an adversary who receives $\mathsf{P1}, \mathsf{P2}$ and $\mathsf{P}_{\mathsf{obf}}$, which is either an obfuscation of $\mathsf{P1}$ or $\mathsf{P2}$. If the adversary receives an obfuscation of $\mathsf{P1}$ then we are in hybrid Hybrid_1 and if the adversary receives the obfuscation of $\mathsf{P2}$ then we are in hybrid Hybrid_2. So, if the adversary could indeed distinguish the two hybrids with non-negligible probability then he can as well distinguish the obfuscations of $\mathsf{P1}$ and $\mathsf{P2}$ with non-negligible probability. This contradicts the differing-inputs property of $\mathcal{P}$ from Corollary 1, thus proving the claim.

Claim 3. Hybrids H_2 and H_3 are computationally indistinguishable under the assumption that the FHE scheme is IND-CPA secure.
Proof. We assume that these two hybrids are distinguishable and then arrive at a contradiction by contradicting the IND-CPA security of the FHE scheme. Suppose there exists an adversary $\mathcal{A}$ that distinguishes hybrids Hybrid_0 and Hybrid_1 then we construct an adversary $\mathcal{A}'$ that breaks the IND-CPA security of FHE scheme as follows. The adversary $\mathcal{A}'$, on input a public key PK_0, first executes $\mathcal{A}$ to get the messages m_0 and m_1. It then sends this to the challenger who decides to encrypt either m_0 or m_1 depending on the challenge bit. The challenge ciphertext, $\mathsf{CT}^{(1)}$, is handed over to $\mathcal{A}'$ who does the following. It generates public key-secret key pair $(\mathsf{SK}_1, \mathsf{PK}_1)$ and then encrypts m_1 using PK_1 to obtain $\mathsf{CT}^{(2)}$. Further, it generates the program $\mathsf{P2}$ in which the decryption is done using the decryption key SK_2. Finally, it computes $\mathcal{P}2_{\mathsf{obf}}$, which is the indistinguishability obfuscation of $\mathsf{P2}$. It gives $(\mathsf{CT}^{(1)}, \mathsf{CT}^{(2)}, \mathsf{P2}_{\mathsf{obf}})$ to $\mathcal{A}$ and then $\mathcal{A}'$ outputs whatever $\mathcal{A}$ outputs.

Claim 4. Hybrids H_3 and H_4 are computationally indistinguishable under the assumption that differing-inputs obfuscators exist for all circuits.
Proof. This is similar to the proof of Claim 2. Consider an adversary who receives $\mathsf{P1}, \mathsf{P2}$ and $\mathsf{P}_{\mathsf{obf}}$, which is either an obfuscation of $\mathsf{P1}$ or $\mathsf{P2}$. If the adversary receives an obfuscation of $\mathsf{P2}$ then we are in hybrid Hybrid_3 and if the adversary receives the obfuscation of $\mathsf{P3}$ then we are in hybrid Hybrid_4. So, if the adversary could indeed distinguish the two hybrids with non-negligible probability then he can as well distinguish the obfuscations of $\mathsf{P1}$ and $\mathsf{P2}$ with non-negligible probability. This contradicts the differing-inputs property of $\mathcal{P}$ from Corollary 1, thus proving the claim.

From the above arguments it follows that hybrids Hybrid_0 and Hybrid_4 are computationally indistinguishable. This proves the differing-inputs of the Turing machines M_0 and M_1. More formally,

Theorem 2. *Under the existence of the following primitives, the construction in Section 3 is a differing-inputs obfuscation for any family of differing-inputs Turing machines.*

- *IND-CPA secure fully homomorphic encryption scheme.*
- *Succinct Non-Interactive Arguments of Knowledge.*
- *Differing-inputs obfuscation for all circuits.*
- *Collision resilient size-reducing hash functions.*

D Proofs of Section 4

Succinctness of functional keys. To argue about the size of a functional key f, we first argue about the size of the program $\mathsf{P}^{(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$ and then we invoke the succinctness property of the differing-inputs obfuscation for Turing machines. This is because, the functional key f is essentially a differing-inputs obfuscation of $\mathsf{P}^{(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$. Let M be the Turing machine implementing the function f. Then, the size of the program is a polynomial in the security parameter, size of M, size of the SNARK verifier and the size of PKE decryptor. Further, the size of the SNARK verifier as well as the PKE decryptor is bounded by a fixed polynomial in the security parameter. Note that here both the SNARK verifier as well as the PKE decryptor are implemented by Turing machines. So, the size of the both the programs is basically a polynomial in the security parameter as well as the size of the Turing machine M. Denote this polynomial by p'. Now, we know that if a Turing machine M' is being obfuscated and its size is s then the size of the obfuscation of M' is $p'(s)$, where p' is also a polynomial. Hence, the size of the obfuscation of $\mathsf{P}^{(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$ is basically $p'(\lambda, |\mathsf{P}^{(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}|)$, and from our earlier observation we have $|\mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}| = p(\lambda, |M|)$. Hence, the size of the functional key is $p''(\lambda, |M|)$ for some fixed polynomial p''.

Input specific running time of decryption algorithm. As in the previous case, we will just argue the running time of the program $\mathsf{P1} = \mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ and then using the bound on the running time of $\mathsf{P1}$ we obtain a bound on the running time of its obfuscation. The running time of the program $\mathsf{P1}$ on input X, is essentially the sum of the running time of the SNARK verifier, the running time of the PKE decryption as well as the running time of M, where M is the Turing machine implementing f. The running time of the SNARK verifier is polynomial in the security parameter and $|X|$. The running time of the PKE decryptor is again a polynomial in the security parameter and $|X|$. Note that $|X|$ is basically a polynomial in $|x|$, where x is the message in contained in both the ciphertexts as part of X. Also, the running time of the Turing machine is $\mathsf{time}(M, x)$, which is at least $|x|$. Overall, the total running time of the obfuscation is a polynomial in the security parameter and $\mathsf{time}(M, |x|)$.

D.1 Proof of Lemma 2

The proof of this lemma is very similar to the proof of Lemma 1 and so we just sketch the details below. Suppose there exists an adversary $\mathcal{A}$ who outputs y such that the output of the programs $\mathsf{P1}$ and $\mathsf{P2}$ are different, where $\mathsf{P1}$ and $\mathsf{P2}$ are the programs output by $\mathsf{Sampler}_{\mathcal{P}}$. Using this adversary, we construct an adversary $\mathcal{A}_{\mathcal{SS}}$ which contradicts the simulation soundness of the SS-NIZK system. To show this, we first recall that all the programs in the family $\mathcal{P}$ have the following three steps in common – SNARK verification phase, PKE decryption and the execution of the function f. Now, if the adversary $\mathcal{A}$ indeed outputs an y, parsed as (z, φ) where φ is a SNARK proof, such that both the programs are different then it has to happen that φ passes the SNARK verification phase. The proof of this follows directly from the proof of Claim 11 in Lemma 1. Now, let z, which is the first component of y be further parsed as $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi)$. We claim that the output of the program $\mathsf{P1}$ (resp., $\mathsf{P2}$) is $M(\mathsf{out}_1)$ (resp., $M(\mathsf{out}_2)$), where out_1 (resp., out_2) is the decryption of CT_1 (resp., CT_2), where M is the Turing machine implementing the function f. This directly follows from the proof of Claim 12 in Lemma 1. Since y is an input such that the output of $\mathsf{P1}$ on input y is different from the output of $\mathsf{P2}$ on input y, we have the fact that $M(\mathsf{out}_1) \neq M(\mathsf{out}_2)$. This can happen only if the message contained in the ciphertexts CT_1 and CT_2 are different. This further means that the SS-NIZK proof contained in the hash h_Π corresponds to a false statement. We use this fact to contradict the simulation soundness of the SS-NIZK proof system as follows. The adversary $\mathcal{A}_{\mathcal{SS}}$, which breaks the simulation soundness of the SS-NIZK proof system, first executes the setup algorithm of PKE system twice to get two pairs of public key-secret keys, namely $(\mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{PK}^2_{\mathsf{PKE}}), (\mathsf{SK}^2_{\mathsf{PKE}}, \mathsf{PK}^2_{\mathsf{PKE}})$. Further it executes the fake setup algorithm of the SNARK proof system to obtain $(\mathsf{CRS}_{\mathsf{SNARK}}, \mathsf{td}_{\mathsf{SNARK}})$. Now, instead of itself executing the setup algoritm of the SS-NIZK proof system it gets the $\mathsf{CRS}_{\mathcal{SS}}$ from the challenger of the SS-NIZK security game. Using the PKE keys, $\mathsf{CRS}_{\mathsf{SNARK}}$ as well as the CRS obtained from the challenger it generates the programs $\mathsf{P1} = \mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$ and $\mathsf{P2} = \mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$. It then passes these two programs to the adversary $\mathcal{A}$ who outputs y. It parses y as $(\mathsf{CT}_1, \mathsf{CT}_2, h_\Pi, \varphi)$. Using the trapdoor of the SNARK extractor it extracts the proof Π corresponding to the instance h_Π. The adversary $\mathcal{A}_{\mathcal{SS}}$ finally outputs $(\mathsf{CT}_1, \mathsf{CT}_2, \Pi)$. Note that if φ is an accepting proof with non-negligible probability then Π is also accepted by the SS-NIZK verifier with non-negligible probability. This fact follows from the extractability property of the SNARK proof system. Further, from our earlier observation $(\mathsf{CT}_1, \mathsf{CT}_2)$ has to be a false statement and hence, Π is a proof for a false statement that is accepted by the SS-NIZK verifier with non-negligible probability, which contradicts the simulation soundness of the SS-NIZK proof system. This completes the proof.

D.2 Proof of indistinguishability of hybrids Hybrid_5 to Hybrid_{11}

We now show that any two consecutive hybrids are computationally indistinguishable from each other thus showing that the hybrids Hybrid_5, which corresponds to the indistinguishability game when x_0, is encrypted and Hybrid_{11}, which corresponds to the indistinguishability game when x_1 is encrypted are computationally indistinguishable from each other. This further proves the security of the FE scheme. We present a sketch of the proofs of the claims since the proofs of the claims that show the indistiguishability of the hybrids are more or less similar to the arguments in Garg et al. [GGH$^+$13b].

Claim 5. There does not exist any PPT adversary who can distinguish the hybrids Hybrid_5 and Hybrid_6 and this follows from the zero knowledge property of our SS-NIZK proof system. [9]
Proof sketch. If there exists a PPT adversary, denoted by $\mathcal{A}^{(5,6)}$, who can distinguish the two hybrids then we construct an adversary, denoted by $\mathcal{A}_{\mathcal{SS}}$, that violates the zero knowledge property of the SS-NIZK proof system. The adversary $\mathcal{A}_{\mathcal{SS}}$ first receives the common reference string $\mathsf{CRS}_{\mathcal{SS}}$ from the challenger (who either uses the honest prover or the simulator). It then executes the keys (public and the secret) of the PKE scheme itself. It then passes the $\mathsf{CRS}_{\mathcal{SS}}$ along with the public keys to the adversary $\mathcal{A}^{(5,6)}$, who then sends the messages x_0 and x_1 to $\mathcal{A}_{\mathcal{SS}}$ who first encrypts a message under both the public keys. It then composes a statement y which says that the encryptions are correctly computed whose witness is essentially the message along with the randomness to generate the encryption. It sends y, along with the witness, to the challenger who produces a proof for the statement. This proof is then sent to $\mathcal{A}_{\mathcal{SS}}$ who, using the PKE ciphertexts along with the proof, composes a FE ciphertext which it sends it to $\mathcal{A}^{(5,6)}$.

If the challenger had used an honest prover to generate the proof then we are in Hybrid_5 else if it used a simulator to generate the proof then we are in Hybrid_6. Hence, if $\mathcal{A}^{(5,6)}$ can distinguish the two hybrids then $\mathcal{A}_{\mathcal{SS}}$ can distinguish the proof produced by the honest prover from the proof produced by the simulator which would contradict the zero knowledge property of $(P_{\mathcal{SS}}, V_{\mathcal{SS}})$.

Claim 6. If our PKE system is IND-CPA secure then there does not exist any PPT adversary who can distinguish the hybrids Hybrid_6 and Hybrid_7 respectively.
Proof sketch. If there exists a PPT adversary $\mathcal{A}^{(6,7)}$ who can distinguish the hybrids Hybrid_6 and Hybrid_7 then we construct an adversary $\mathcal{A}_{\mathsf{PKE}}$ who can violate the security of the PKE scheme. Adversary executes the setup algorithm and sends the public parameters to the adversary $\mathcal{A}^{(6,7)}$. Just like the previous hybrid, even in this hybrid the adversary $\mathcal{A}^{(6,7)}$ generates the fake CRS for the SS-NIZK proof system. The adversary $\mathcal{A}_{\mathsf{PKE}}$ then obtains the messages x_0 and x_1 from $\mathcal{A}^{(6,7)}$ which it then sends to the challenger. The challenger will encrypt either x_0 or x_1 and then sends the encryption to $\mathcal{A}_{\mathsf{PKE}}$. Then, $\mathcal{A}_{\mathsf{PKE}}$ executes the setup algorithm of the PKE scheme to obtain a new public key-secret key pair. It then encrypts x_1 using the new public key. It then generates $\mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}}$ corresponding to the common reference strings of the SS-NIZK and the SNARK system respectively. It then generates the proofs as described in the FE encryption algorithm. Finally, it sends the FE ciphertext to the adversary $\mathcal{A}^{(6,7)}$. The output of $\mathcal{A}$ determines the output of $\mathcal{A}_{\mathsf{PKE}}$.

If the challenger gave an encryption of x_0 to $\mathcal{A}_{\mathsf{PKE}}$ then we are in Hybrid_6 and if the challenger gave an encryption of x_1 then we are in Hybrid_7. Hence, if $\mathcal{A}^{(6,7)}$ can distinguish both the hybrids then the adversary $\mathcal{A}_{\mathsf{PKE}}$ can violate the security of the IND-CPA scheme.

[9] The main difference between this proof and the proof in Garg et al. [GGH$^+$13b] is that in their case, CRS had to be produced by the simulator after the message was fixed whereas in our case, CRS can be generated by the simulator even before the messages are fixed. This is precisely the reason why we are able to achieve full security whereas their construction achieves only selective security.

Claim 7. If the differing-inputs assumption holds for the family $\mathcal{P}$ then for every $i \in [0, q-1]$, there does not exist any PPT adversary that can distinguish the hybrids $\mathsf{Hybrid}_{8,i}$ and $\mathsf{Hybrid}_{8,i+1}$.
Proof. Suppose there exists an adversary $\mathcal{A}_8^{(i,i+1)}$ that can distinguish the hybrids $\mathsf{Hybrid}_{8,i}$ and $\mathsf{Hybrid}_{8,i+1}$ then we construct an adversary $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ who violates the differing-inputs property of $\mathcal{P}$ as follows. The challenger enerates both the public keys of the PKE scheme as well as the common reference strings $\mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}}$ of the SS-NIZK as well as the SNARK system respectively all by itself. Like in the previous hybrid, the common reference string of the SS-NIZK proof system, namely $\mathsf{CRS}_{\mathcal{SS}}$ is simulated here. It then passes the public parameters, which include the public keys as well as the common reference strings, to $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ who in turn sends it to $\mathcal{A}_{8,i+1}$.

$\mathcal{A}_8^{(i,i+1)}$ then makes the key queries to $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ which it forwards to the challenger. For $j \leq i$, the private key is generated as an obfuscation of the program $\mathsf{P1} = \mathsf{P}^{(f,\mathsf{SK}^1_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$. And for $j > i+1$, the j^{th} private key is created as an obfuscation of the program $\mathsf{P2} = \mathsf{P}^{(f,\mathsf{SK}^2_{\mathsf{PKE}},\mathsf{CRS}_{\mathcal{SS}},\mathsf{CRS}_{\mathsf{SNARK}})}$. For the $i+1^{th}$ private key query, the challenger either chooses to obfuscate the program $\mathsf{P1}$ or $\mathsf{P2}$, denoted by $\mathsf{P}_{\mathsf{obf}}$, which it sends to the adversary $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ which forwards it to $\mathcal{A}_8^{(i,i+1)}$. The output of $\mathcal{A}_8^{(i,i+1)}$ is essentially the output of $\mathcal{A}_{\mathsf{di}\mathcal{O}}$.

Note that the distribution to generate the above programs which are submitted to the challenger for obfuscation is identical to the output distribution of the sampler algorithm of $\mathcal{P}$. And hence, invoking the differing-inputs obfuscation on the program family $\mathcal{P}$ we get the fact that the obfuscations of both the programs are computationally indistinguishable from each other.

If the challenger in the differing-inputs security game chose $\mathsf{P1}$ then we are in hybrid $\mathsf{Hybrid}_{8,i}$ and if it chose $\mathsf{P2}$, then we are in $\mathsf{Hybrid}_{8,i+1}$. And so, if an adversary $\mathcal{A}_8^{i,i+1}$ can distinguish between the two hybrids with non-negligible probability then it will violate the fact that the obfuscations of both the programs are computationally indistinguishable from each other.

Claim 8. If our PKE system is IND-CPA secure then no PPT adversary can distinguish with non-negligible probability between $\mathsf{Hybrid}_{8,q}$ and Hybrid_9.
Proof. The proof of the above claim is similar to the proof of Claim 2. If there exists a PPT adversary $\mathcal{A}^{(8,9)}$ who can distinguish the hybrids $\mathsf{Hybrid}_{8,q}$ and Hybrid_9 then we construct an adversary $\mathcal{A}_{\mathsf{PKE}}$ who can violate the security of the PKE scheme. Adversary executes the setup algorithm and sends the public parameters to the adversary $\mathcal{A}^{(8,9)}$. Just like the previous hybrid, even in this hybrid the adversary $\mathcal{A}^{(8,9)}$ generates the fake CRS for the SS-NIZK proof system. The adversary $\mathcal{A}_{\mathsf{PKE}}$ then obtains the messages x_0 and x_1 from $\mathcal{A}^{(8,9)}$ which it then sends to the challenger. The challenger will encrypt either x_0 or x_1 and then sends the encryption to $\mathcal{A}_{\mathsf{PKE}}$. Then, $\mathcal{A}_{\mathsf{PKE}}$ executes the setup algorithm of the PKE scheme to obtain a new public key-secret key pair. It then encrypts x_0 using the new public key. As in the previous hybrids, we generate the simulated SS-NIZK proof. Finally, it sends the FE ciphertext to the adversary $\mathcal{A}^{(8,9)}$. The output of $\mathcal{A}$ determines the output of $\mathcal{A}_{\mathsf{PKE}}$.

If the challenger gave an encryption of x_0 then we are in $\mathsf{Hybrid}_{8,q}$ and if the challenger gave an encryption of x_1 then we are in Hybrid_9. Hence, if $\mathcal{A}^{(8,9)}$ can distinguish both the hybrids then the adversary $\mathcal{A}_{\mathsf{PKE}}$ can violate the security of the IND-CPA scheme.

Claim 9. If the differing-inputs assumption holds for the program family $\mathcal{P}$ then no PPT adversary can distinguish between $\mathsf{Hybrid}_{10,i}$ and $\mathsf{Hybrid}_{10,i+1}$ for $i \in [0, q-1]$.
Proof. The proof of the above claim is similar to the proof of Claim 5. Suppose there exists an adversary $\mathcal{A}_{10}^{(i,i+1)}$ that can distinguish the hybrids $\mathsf{Hybrid}_{10,i}$ and $\mathsf{Hybrid}_{10,i+1}$ then we construct an adversary $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ who violates the differing-inputs property of $\mathcal{P}$ as follows. The challenger enerates both the public keys of the PKE scheme as well as the common reference strings

$\mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}}$ of the SS-NIZK as well as the SNARK system respectively all by itself. Like in the previous hybrid, the common reference string of the SS-NIZK proof system, namely $\mathsf{CRS}_{\mathcal{SS}}$ is simulated here. It then passes the public parameters, which include the public keys as well as the common reference strings, to $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ who in turn sends it to $\mathcal{A}_{10,i+1}$.

$\mathcal{A}_{10}^{(i,i+1)}$ then makes the key queries to $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ which it forwards to the challenger. For $j \leq i$, the private key is generated as an obfuscation of the program $\mathsf{P1} = \mathsf{P}^{(f, \mathsf{SK}^1_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$. And for $j > i+1$, the j^{th} private key is created as an obfuscation of the program $\mathsf{P2} = \mathsf{P}^{(f, \mathsf{SK}^2_{\mathsf{PKE}}, \mathsf{CRS}_{\mathcal{SS}}, \mathsf{CRS}_{\mathsf{SNARK}})}$. For the $i+1^{th}$ private key query, the challenger either chooses to obfuscate the program $\mathsf{P1}$ or $\mathsf{P2}$, denoted by $\mathsf{P}_{\mathsf{obf}}$, which it sends to the adversary $\mathcal{A}_{\mathsf{di}\mathcal{O}}$ which forwards it to $\mathcal{A}_{10}^{(i,i+1)}$. The output of $\mathcal{A}_{8}^{(i,i+1)}$ is essentially the output of $\mathcal{A}_{\mathsf{di}\mathcal{O}}$.

Note that the distribution to generate the above programs which are submitted to the challenger for obfuscation is identical to the output distribution of the sampler algorithm of $\mathcal{P}$. And hence, invoking the differing-inputs obfuscation on the program family $\mathcal{P}$ we get the fact that the obfuscations of both the programs are computationally indistinguishable from each other.

If the challenger in the differing-inputs security game chose $\mathsf{P2}$ then we are in hybrid $\mathsf{Hybrid}_{10,i}$ and if it chose $\mathsf{P1}$, then we are in $\mathsf{Hybrid}_{10,i+1}$. And so, if an adversary $\mathcal{A}_8^{i,i+1}$ can distinguish between the two hybrids with non-negligible probability then it will violate the fact that the obfuscations of both the programs are computationally indistinguishable from each other.

Claim 10. If our SS-NIZK system is computational zero knowledge then no PPT distinguisher with non-negligible probability can distinguish the hybrids $\mathsf{Hybrid}_{10,q}$ and Hybrid_{11}, for $i \in [0, q]$.
Proof. If there exists a PPT adversary, denoted by $\mathcal{A}^{(10,11)}$, who can distinguish the two hybrids then we construct an adversary, denoted by $\mathcal{A}_{\mathcal{SS}}$, that violates the zero knowledge property of the SS-NIZK proof system. The adversary $\mathcal{A}_{\mathcal{SS}}$ first receives the common reference string $\mathsf{CRS}_{\mathcal{SS}}$ from the challenger (who either uses the honest prover or the simulator). It then executes the keys (public and the secret) of the PKE scheme itself. It then passes the $\mathsf{CRS}_{\mathcal{SS}}$ along with the public keys to the adversary $\mathcal{A}^{(10,11)}$, who then sends the messages x_0 and x_1 to $\mathcal{A}_{\mathcal{SS}}$ who first encrypts a message under both the public keys. It then composes a statement y which says that the encryptions are correctly computed whose witness is essentially the message along with the randomness to generate the encryption. It sends y, along with the witness, to the challenger who produces a proof for the statement. This proof is then sent to $\mathcal{A}_{\mathcal{SS}}$ who, using the PKE ciphertexts along with the proof, composes a FE ciphertext which it sends it to $\mathcal{A}^{(10,11)}$.

If the challenger had used an honest prover to generate the proof then we are in Hybrid_{11} else if it used a simulator to generate the proof then we are in $\mathsf{Hybrid}_{10,q}$. Hence, if $\mathcal{A}^{(10,11)}$ can distinguish the two hybrids then $\mathcal{A}_{\mathcal{SS}}$ can distinguish the proof produced by the honest prover from the proof produced by the simulator which would contradict the zero knowledge property of $(P_{\mathcal{SS}}, V_{\mathcal{SS}})$.

Theorem 3. *Under the following assumptions we have the fact that the functional encryption system constructed in Section 4 is fully secure according to the indistinguishability game described in Section A.2.*

- *An IND-CPA secure public key encryption scheme exists.*
- *A simulation-sound NIZK proof system exists.*
- *Succinct Non Interactive Arguments of Knowledge exists.*
- *Differing-inputs obfuscation for Turing machines exists.*
- *Collision resilient size-reducing hash functions.*

Proof. The above claims show that every consecutive hybrids are computationally indistinguishable. This means that the hybrids Hybrid_5 and Hybrid_{11} are computationally indistinguishable.

This proves the security of the indisitnguishability game because Hybrid_5 corresponds to the indisinguishability when the message x_0 is encrypted and hybrid Hybrid_{11} corresponds to the indistinguishability game when the message x_1 is encrypted. $\square$