Functional Encryption for Turing Machines

Prabhanjan Ananth* and Amit Sahai**

Department of Computer Science and Center for Encrypted Functionalities,
University of California, Los Angeles,
USA

Abstract. In this work, we construct an adaptively secure functional encryption for Turing machines scheme, based on indistinguishability obfuscation for circuits. Our work places no restrictions on the types of Turing machines that can be associated with each secret key, in the sense that the Turing machines can accept inputs of unbounded length, and there is no limit to the description size or the space complexity of the Turing machines.

Prior to our work, only special cases of this result were known, or stronger assumptions were required. More specifically, previous work (implicitly) achieved selectively secure FE for Turing machines with a-priori bounded input based on indistinguishability obfuscation (STOC 2015), or achieved FE for general Turing machines only based on knowledge-type assumptions such as public-coin differing-inputs obfuscation (TCC 2015).

A consequence of our result is the first constructions of *succinct* adaptively secure garbling schemes (even for circuits) in the standard model. Prior succinct garbling schemes (even for circuits) were only known to be adaptively secure in the random oracle model.

1 Introduction

Contemporary cloud-based computing systems demand encryption schemes that go far beyond the traditional goal of merely securing a communication channel. The notion of functional encryption, first conceived under the name of Attribute-Based Encryption in [**?**] and formalized later in the works of [**?**,**?**], has emerged as a powerful form of encryption well-suited to many contemporary applications (see [**?**,**?**] for further discussion of application scenarios for functional encryption). A functional encryption (FE) scheme allows a user possessing a key associated

* Email: prabhanjan.va@gmail.com. This work was partially supported by grant #360584 from the Simons Foundation.

** Email: sahai@cs.ucla.edu. Research supported in part from a DARPA/ARL SAFEWARE award, NSF Frontier Award 1413955, NSF grants 1228984, 1136174, 1118096, and 1065276, a Xerox Faculty Research Award, a Google Faculty Research Award, an equipment grant from Intel, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through the ARL under Contract W911NF-15-C-0205. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

with a function f to recover the output $f(x)$, given an encryption of x. The intuitive security guarantee of a FE scheme dictates that the only information about x revealed to the user is $f(x)$. Furthermore, if the user obtains keys for many functions $f_1, \ldots f_k$, then the user should only learn $f_1(x), \ldots, f_k(x)$ and nothing more. It turns out that formalizing security using a simulation-based definition leads to impossibility results [**?**,**?**]; however, there are sound adaptive indistinguishability-based formulations [**?**] that also imply simulation-based security in restricted settings [**?**]. Following most recent work on FE [**?**,**?**,**?**,**?**], we will focus on achieving this strong indistinguishability-based notion of security here.
In this work, we address the following basic question:

"Is FE possible for functions described by arbitrary Turing machines?"

Previous work and its limitations. There have been many works on functional encryption over the past few years but a satisfying answer to this question has remained elusive.
The first constructions of FE considered only limited functions, such as inner product [**?**]. The first constructions of FE that allowed for more general functions considered the setting where the adversary can just request a single (or a bounded number of) key queries [**?**,**?**], but only for functions represented by circuits. A major advance occurred in the work of [**?**], which constructed an FE scheme allowing for functions specified by arbitrary circuits, with no bound on key queries, based on indistinguishability obfuscation (iO) for circuits. Since this work, the assumption of iO for circuits has become the staple assumption in this area.
However, [**?**] and other FE results deal with functionalities represented by *circuits* – and representing functions as circuits gives rise to two major drawbacks. The first drawback is that a circuit representation takes the worst case running time on every input. Research to deal with this issue was initiated by Goldwasser et al. [**?**], and there have been several recent works [**?**,**?**,**?**,**?**], that (implicitly or explicitly) give rise to FE schemes with input-specific runtimes based on iO for circuits.
The second drawback is that the input length of the function is a-priori bounded. In many scenarios, especially involving large datasets, having an a-priori bound is clearly unreasonable. For example, if functional encryption is used for allowing a researcher to perform some data analysis on hospital records, then having a bound on input length would require that there be an a-priori bound, at the time of setting up the encryption scheme, on the length of encrypted hospital records, which seems quite unreasonable. In general, we would like to represent the function being computed as a Turing Machine, that can accept inputs of arbitrary length. The problem of constructing FE schemes which can handle messages of unbounded length has remained largely open: the recent works of [**?**,**?**,**?**] construct iO for Turing Machines only with bounded input length, where the bound must be specified at the time of obfuscating the Turing Machine. If this iO method is combined, for example, with the FE construction recipe of [**?**], then this would only yield FE for functions with a bound on input length specified at the time of setting up the FE scheme.

There have been works [**?**,**?**] on overcoming the issue of a priori bounded input lengths but these are based on strong knowledge-type assumptions called differing inputs obfuscation [**?**,**?**,**?**] or more recently public-coin differing inputs obfuscation [**?**]. Our main contribution is developing new technical approaches that allow us to remove the need for such assumptions, and use only iO for circuits[1].

Results and Technical Overview. We prove the following informal theorem.

Theorem 1 (Informal). *There exists a public-key FE scheme, assuming the existence of indistinguishability obfuscation and one-way functions, that satisfies the following properties:*

1. *There is no a priori bound on the number of functional keys issued.*
2. *The secret keys correspond to Turing machines.*
3. *It achieves adaptive security.*
4. *There is no a priori bound on length of the plaintext and the size of the Turing machine.*
5. *The running time of encryption is independent of the Turing machine size. The running time of the key generation is independent of the plaintext size.*

A corollary of the above theorem is the first construction of succinct adaptively secure garbling schemes for TMs (with indistinguishability-based security) in the standard model. By succinctness, we mean that the size of the input encoding is independent of the function (circuit or TM) size. Prior solutions were either shown in the random oracle model [**?**,**?**] or under restricted settings [**?**].

We now give a roadmap for the overall approach and the techniques we use to achieve our result.
To gather some ideas towards achieving our goal of adaptive FE for TMs, we first focus on the simplest possible scenario of FE for Turing machines: adversary can make only a single ciphertext query and a function query, and furthermore we work in the secret-key setting. We call a FE scheme satisfying this security notion to be 1-CT 1-Key Private-key FE.

Initial goal: Adaptive 1-CT 1-Key Private-key FE for TMs. To build an adaptive 1-CT 1-key private-key FE for TMs scheme, we first take inspiration from the corresponding FE for *circuits* constructions known in the literature to see what tools might be helpful here. Sahai and Seyalioglu [**?**] and Gorbunov et al. [**?**] give constructions using the tool of randomized encodings (RE) of computation. A randomized encoding is a representation of a function along with an input that is simpler to compute than the function itself. Further this representation reveals only the output of the function and nothing else. In other words, given functions f_1, f_2 and inputs x_1, x_2 such that $f_1(x_1) = f_2(x_2)$, it should be the case that the encoding of (f_1, x_1) should be computationally indistinguishable from an encoding of (f_2, x_2). Such randomized encodings for TMs were recently constructed in [**?**,**?**,**?**], based on iO for circuits.

[1] We stress that despite recent cryptanalytic progress, iO candidates such as [**?**] remain beyond the reach of any known cryptanalytic technique.

The essential difference between a randomized encoding and what we need for a 1-CT 1-key FE scheme concerns two additional features that we would need from the randomized encoding:

- First, we need the randomized encoding to be computable *separately* for the function and the input. That is, given only f, it should be possible to compute an encoding $\hat{f}$; and given only x, it should be possible to compute an encoding $\hat{x}$; such that $(\hat{f}, \hat{x})$ constitute a randomized encoding of (f, x). We need this because the ciphertext will be akin to the encoding of the input, whereas the private key will be akin to the encoding of the function. This is essentially the notion of a decomposable randomized encoding [**?**].
- Then, more crucially, we also need to strengthen our notion of security: In a standard randomized encoding scheme, the adversary needs to declare f_1, f_2, x_1, x_2 all at the beginning, and then we have the guarantee that $(\hat{f_1}, \hat{x_1})$ is computationally indistinguishable to $(\hat{f_2}, \hat{x_2})$. However, for an FE scheme, even with just "selective" security, the adversary is given the power to adaptively specify at least f_1, f_2 after it has seen the encodings $\hat{x_1}$ and $\hat{x_2}$. More generally, we would like to have security where the adversary can choose whether it would like to specify f_1, f_2 first or x_1, x_2 first.

It turns out that achieving these two properties is relatively straightforward when dealing with randomized encodings of circuits using Yao's garbled circuits [**?**]. It is not so straightforward for us in the context of TMs and adaptive security, as we explain below.

To see why our situation is nontrivial and to get intuition about the obstacles we must overcome, let us first consider a *failed attempt* to achieve these properties by trying to apply the generic transformation, which was formalized in the work of Bellare et al. [**?**], to achieve adaptive security: in this attempt, the new input encoding and new function encoding will now be $(\hat{x} \oplus R, S)$ and $(R, \hat{f} \oplus S)$, respectively, where R and S are random strings. The idea behind this transformation is as follows: no matter what the adversary queries for (input or function) in the beginning, it is just given two random strings (R, S). When the adversary makes the other query, the simulator would know at this point both the input and the function. Hence, it would obtain the corresponding encodings $\hat{f}$ and $\hat{x}$ from the ordinary security of the randomized encoding scheme. Now, the simulator would respond to the adversary by giving $(\hat{x} \oplus R, \hat{f} \oplus S)$ thus successfully simulating the game. The problem with this solution for us lies in the *sizes* of the encodings. If we look at the strings R and S, they are as long as the length of $\hat{x}$ and $\hat{f}$ respectively. This would mean that the size of the new input encoding (resp., new function encoding) depends on the function length (resp., input length) – which violates our main goal of achieving FE without restrictions on input length!

Revisiting the KLW randomized encoding. In order to achieve our goal, we will need to look at the specifics of the decomposable RE for TMs construction in [**?**]. We then develop new ideas specific to the construction that help us achieve adaptive security. Before we do that, we revisit the KLW randomized encoding at a high level, sufficient for us to explain the new ideas in our work. The encoding procedure of a Turing machine M and input x consists of the following two main steps:

1. The storage tape of the TM is initialized with the encryption of x. It then builds an accumulator storage tree on the ciphertext. The accumulator storage tree resembles a Merkle hash tree with the additional property that this tree is unconditionally sound for a select portion of the storage. The root of the tree is then authenticated.
2. A program that computes the next step function of the Turing machine M is then designed. This program enables computation of M one step at a time. This program has secrets that enable decrypting encrypted tape symbols and also to perform some checks on the input encrypted symbol. To hide the secrets, this program is obfuscated.

The decoding just involves running the next message function repeatedly on the computation obtained so far until the Turing Machine terminates. At this point, the decode algorithm will output whatever the Turing Machine outputs.

First Step towards Adaptivity: 3-Stage KLW. The main issue with trying to use the random masking technique was that we were trying to use randomness to mask the entire input encoding or the function encoding, which could be of unbounded length. So our main goal will be to find a way to achieve adaptivity where randomness need only be used to mask *bounded* portions of the encoding.

As a first step towards achieving this, we want to symmetrize how we treat the input x and the function f. We do this by treating both x and f as being inputs to a Universal Turing Machine U, where U is both of bounded size and is entirely known a-priori, such that $U(f, x) = f(x)$. That is, we have three algorithms[2]: InpEnc outputs an encoding of input x, FnEnc outputs an encoding of f, and UTMEnc outputs a TM encoding of UTM.

A natural approach would be to try to use the KLW scheme sketched above to achieve the goal. The only difference is that, unlike the original KLW scheme, in the 3-stage KLW scheme, the input encoding is split into two encodings (InpEnc and FnEnc) and so there must be a way to stitch the input encodings into one. We develop a mechanism, called $\mathsf{combiner}$, to achieve this goal. A combiner is an algorithm that combines two input encodings into one input encoding. Furthermore, the combiner algorithm we develop is succinct; it only takes a portion of the two encodings (of say, x and f) and spits out an element that together with the encodings of x and f represent $x||f$. Note, however, that the combiner algorithm needs secret information in order to perform its combining role correctly.

The key to constructing this combiner is the accumulator storage scheme of KLW. Recall that the accumulator storage on $(x||f)$ was essentially a binary tree on $x||f$. We modify this accumulator storage such that the storage tree on $(x||f)$ can be built by first building a storage tree on x, then building a separate independent storage tree on f, and then joining both these two trees by making them children of a root node. Once we have this tool, developing our combiner algorithm is easy: the input encoding of x consists of a storage tree on an encryption of x, encoding of f consists of a storage tree on the encryption of f. The combine algorithm

[2] The actual algorithms as presented in the technical section is slightly different. We chose to present it this way in the introduction for intuitive clarity.

then takes *only* the root nodes of both these two trees and creates a new root node which is the parent of these two root nodes. The combiner then signs on the root node as a means of authenticating the fact that this new root node was created legally.

We are almost ready to now apply the random masking technique to achieve adaptive security by masking our new succinct representations. However, there is a problem: the combiner algorithm. In 3-stage KLW, once we have encodings of x and f, before we can have a randomized encoding, these two encodings need to be combined using secret information. This is not allowed in a randomized encoding, where the decode algorithm must be public.

Getting rid of combiner: 2-ary FE for TMs (1-CT 1-Key setting). Since we need to eliminate the need for the combiner algorithm, we start by trying to delegate the combine operation to the decoder. We can attempt to do so by including an obfuscated version of the combiner program as part of the encoding itself, where obfuscation is needed since the combiner procedure contains some secret values that have to be hidden. By itself, however, this approach does not work, because the adversary who now possesses the obfuscated combine program can now illegally combine different storages (other than those corresponding to x and f) – we term this type of attack as a *mixed storage attack*.

To prevent mixed storage attacks, we use splittable signatures: the challenger can sign the root of the storage of x as well as the root of the storage of f. The obfuscated program now only outputs the combined value if the signatures can be verified correctly. By using splittable signatures, we can argue that the adversary is prevented from mixed storage attacks relying only on indistinguishability obfuscation for circuits.

Once we have the obfuscated combiner program, the next issue is whether the obfuscated combiner should be included as part of InpEnc or FnEnc. Including it in either of them will cause problems because the simulator needs to simulate the appropriate parameters in the combiner algorithm and it can do that only after looking at both the InpEnc and FnEnc queries. Here we can (finally!) apply the random masking technique since the size of the combiner is independent of the size of the input as well as the function and thus the length of the random mask needed is small. The resulting scheme that we get is a 2-ary FE [**?**] for TMs, where the adversary can only make a single message and key query – note that it is essentially the same as 3-stage KLW scheme except that it does not have the combiner algorithm.

Using some additional but similar ideas, we can show that the algorithms FnEnc and UTMEnc can be combined into one encoding. The result is a scheme with an input encoding, function encoding and a decode algorithm with the security guarantee that the input query and the function query can be made adaptively, which is precisely the goal we had started off with.

Boosting mechanism: 1-Key 1-CT (private-key) FE to many-key (public-key) FE. Now that we have achieved the goal of single-ciphertext single-key private key FE for TMs, the next direction is to explore whether there is any way to combine this with other known tools

to obtain a public-key FE with unbounded number of function queries. We give a mechanism of combining the 1-Key 1-CT FE scheme with other FE schemes that are defined for *circuits* to obtain a public-key FE scheme for Turing machines. Further, our resulting FE scheme is such that it is adaptively secure assuming only that the 1-Key 1-CT FE scheme is adaptively secure. The high level approach is that the ciphertexts and the functional keys are designed such that every ciphertext-functional key pair gives rise to a unique instantiation of single-ciphertext single-key private FE. This is reminiscent of the approach of Waters [**?**], later revisited by [**?**], in the context of constructing adaptively secure FE for circuits.

Our boosting mechanism, however, diverges in several ways from the previous works of [**?**,**?**]. First, we note that just syntactically, our boosting mechanism is the first such mechanism that uses only 1-Key 1-CT FE as a building block; in contrast, for example, [**?**] needed *many*-Key 1-CT FE as a building block.

Zooming in on the main new idea we develop for our boosting mechanism, we find that it is used exactly to deal with the fact that unbounded inputs that must be embedded in ciphertexts. Note that all previous FE schemes placed an a-priori bound on the inputs to be encrypted in ciphertexts. Therefore, to build our encryption mechanism, we cannot use previous FE encryption to encode inputs. We also cannot directly use the 1-Key 1-CT FE, since this scheme can only support a single key and a single ciphertext. To resolve this dilemma, we note that even though previous FE schemes could not handle inputs of unbounded length, previous FE schemes can handle *keys* corresponding to arbitrary-length circuits. Therefore, crucially in our boosting procedure, when encrypting an input x, we actually prepare a circuit H_x that has x built into it, and then use an existing FE scheme to prepare a key corresponding to H_x. Here we make use of the Brakerski-Segev [**?**] transformation to guarantee that the key for H_x does not leak x. We utilize a new layer of indirection, where this circuit H_x expects to receive as input the master secret key of a 1-Key 1-CT FE scheme, and then uses this master secret key to create a 1-Key 1-CT encryption of x. In this way, the final FE scheme that we construct inherits the security of the 1-Key 1-CT encryption scheme, but a fresh and independent instance of the 1-Key 1-CT scheme is created for each pair of (input, function) that is ever considered within our final FE scheme.

Subsequent Work. Recently, Nimishaki, Wichs and, Zhandry [**?**] construct a traitor tracing scheme which allows for embedding user information in the issued keys. One of the main tools used to construct this primitive is an adaptively secure FE scheme. As a first step, they show how to achieve a traitor tracing scheme from a private linear broadcast encryption (PLBE) scheme defined for a large identity space. In the next step, they show how to design a PLBE scheme from adaptive FE.

2 Preliminaries

We denote λ to be the security parameter. We say that a function $\mu(\lambda)$ is negligible if for any polynomial $p(\lambda)$ it holds that $\mu(\lambda) < 1/p(\lambda)$ for all sufficiently large $\lambda \in \mathbb{N}$. We use the notation negl to denote a negligible function.

We assume that the reader is familiar with the notion of Turing machines, standard cryptographic notions of pseudorandom functions and symmetric encryption schemes. We use the convention that a Turing machine also outputs the time it takes to execute. As a consequence, if we have $M_0(x) = M_1(x)$ then it means that not only are the outputs same but even the running times are the same.

2.1 Functional Encryption for Turing machines

We now define the notion of functional encryption (FE) for Turing machines. This notion differs from the traditional notion of FE for circuits (to be defined later) in that the functional keys are associated to Turing machines as against circuits. Further, the functional keys can be used to decrypt ciphertexts of messages of arbitrary length and the decryption time depends only the running time of the Turing machine on the message.

A public-key functional encryption scheme, defined for a message space $\mathcal{M}$ and a class of Turing machines $\mathcal{F}$, consists of four PPT algorithms $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ described as follows.

- $\mathsf{Setup}(1^\lambda)$: The setup algorithm takes as input the security parameter λ in unary and outputs a public key-secret key pair $(\mathsf{PK}, \mathsf{MSK})$.
- $\mathsf{KeyGen}(\mathsf{MSK}, f \in \mathcal{F})$: The key generation algorithm takes as input the master secret key MSK, a Turing machine $f \in \mathcal{F}$ [3], and outputs a functional key sk_f.
- $\mathsf{Enc}(\mathsf{PK}, m \in \mathcal{M})$: The encryption algorithm takes as input the public key PK, a message $m \in \mathcal{M}$ and outputs a ciphertext CT.
- $\mathsf{Dec}(sk_f, \mathsf{CT})$: The decryption algorithm takes as input the functional key sk_f, a ciphertext CT and outputs $\hat{m}$.

The FE scheme defined above, in addition to correctness and security, needs to satisfy the efficiency property. All these properties are defined below.

Correctness. The correctness notion of a FE scheme dictates that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, for every message $m \in \mathcal{M}$, and for every Turing machine $f \in \mathcal{F}$ it holds that $\Pr\left[f(m) \leftarrow \mathsf{Dec}(\mathsf{KeyGen}(\mathsf{MSK}, f), \mathsf{Enc}(\mathsf{PK}, m))\right] \geq 1 - \mathsf{negl}(\lambda)$, where $(\mathsf{PK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

[3] We use the same notation to denote the function as well as the Turing machine representing the function f.

Efficiency. The efficiency property of a public-key FE scheme says that the algorithm Setup on input 1^λ should run in time polynomial in λ, KeyGen on input the Turing machine f (along with master secret key) should run in time polynomial in $(\lambda, |f|)$, Enc on input a message m (along with the public key) should run in time polynomial in $(\lambda, |m|)$. Finally, Dec on input a functional key of f and an encryption of m should run in time polynomial in $(\lambda, |f|, |m|, \mathsf{timeTM}(f, m)))$.

Security. The security notion we define is identical to the indistinguishability-based security notion defined for circuits.

Definition 1. *A public-key functional encryption scheme* $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *over a class of Turing machines* $\mathcal{F}$ *and a message space* $\mathcal{M}$ *is* ***adaptively secure*** *if for any PPT adversary* $\mathcal{A}$ *there exists a negligible function* $\mu(\lambda)$ *such that for all sufficiently large* $\lambda \in \mathbb{N}$*, the advantage of* $\mathcal{A}$ *is defined to be*

$$\mathsf{Adv}^{\mathsf{FE}}_{\mathcal{A}} = \left|\mathsf{Prob}[\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, 0) = 1] - \mathsf{Prob}[\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda),$$

where for each $b \in \{0, 1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, b)$*, modeled as a game between the challenger and the adversary* $\mathcal{A}$*, is defined as follows:*

1. *The challenger first executes* $\mathsf{Setup}(1^\lambda)$ *to obtain* $(\mathsf{PK}, \mathsf{MSK})$*. It then sends* PK *to the adversary.*
2. ***Query Phase I****: The adversary submits a Turing machine query* f *to the challenger. The challenger sends back* sk_f *to the adversary, where* sk_f *is the output of* $\mathsf{KeyGen}(\mathsf{MSK}, f)$*.*
3. ***Challenge Phase****: The adversary submits a message-pair* (m_0, m_1) *to the challenger. The challenger checks whether* $f(m_0) = f(m_1)$ *for all Turing machine queries* f *made so far. If this is not the case, the challenger aborts. Otherwise, the challenger sends back* $\mathsf{CT} = \mathsf{Enc}(\mathsf{MSK}, m_b)$*.*
4. ***Query Phase II****: The adversary submits a Turing machine query* f *to the challenger. The challenger generates* sk_f*, where* sk_f *is the output of* $\mathsf{KeyGen}(\mathsf{MSK}, f)$*. It sends* sk_f *to the adversary only if* $f(m_0) = f(m_1)$*, otherwise it aborts.*
5. *The output of the experiment is* b'*, where* b' *is the output of* $\mathcal{A}$*.*

We can also consider a weaker notion, termed as *selective security*, where the adversary has to submit the challenge message pair at the beginning of the game itself even before it receives the public parameters and such a FE scheme is said to be *selectively secure*.

Private Key Setting. We can analogously define the notion of FE for TMs in the private-key setting. The difference between the public-key setting and the private-key setting is that in the private-key setting, the encryptor needs to know the master secret key to encrypt the messages. We provide the formal definition of private-key FE for TMs in the full version [**?**].

2.2 (Compact) FE for circuits

Public-Key FE One of the building blocks in our construction of FE for TMs is a public-key FE for circuits (i.e., the functions are represented as circuits). We now recall its definition from [**?**,**?**].
A public-key functional encryption (FE) scheme PubFE, defined for a class of functions $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda\in\mathbb{N}}$ and message space $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$, is represented by four PPT algorithms, namely (Setup, KeyGen, Enc, Dec). The input length of any $f \in \mathcal{F}_\lambda$ is the same as the length of any $m \in \mathcal{M}_\lambda$. The description of these four algorithms is given below.

- $\mathsf{Setup}(1^\lambda)$: It takes as input a security parameter λ in unary and outputs a public key-secret key pair $(\mathsf{PK}, \mathsf{MSK})$.
- $\mathsf{KeyGen}(\mathsf{MSK}, f \in \mathcal{F}_\lambda)$: It takes as input a secret key MSK, a function $f \in \mathcal{F}_\lambda$ and outputs a functional key sk_f.
- $\mathsf{Enc}(\mathsf{PK}, m \in \mathcal{M}_\lambda)$: It takes as input a public key PK, a message $m \in \mathcal{M}_\lambda$ and outputs an encryption of m.
- $\mathsf{Dec}(sk_f, \mathsf{CT})$: It takes as input a functional key sk_f, a ciphertext CT and outputs $\widehat{m}$.

We require the FE scheme to satisfy the efficiency property in addition to the traditional properties of correctness and security.

Correctness. The correctness property says that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, for every message $m \in \mathcal{M}_\lambda$, and for every function $f \in \mathcal{F}_\lambda$ it holds that $\Pr[f(m) \leftarrow \mathsf{Dec}(\mathsf{KeyGen}(\mathsf{MSK}, f), \mathsf{Enc}(\mathsf{PK}, m))] \geq 1 - \mathsf{negl}(\lambda)$, where $(\mathsf{PK}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Efficiency. At a high level, the efficiency property says that the setup and the encryption algorithm is independent of the size of the circuits for which functional keys are produced. More formally, the running time of the setup algorithm, $\mathsf{Setup}(1^\lambda)$ is a polynomial in just the security parameter λ and the encryption algorithm, $\mathsf{Enc}(\mathsf{PK}, m)$ is a polynomial in only the security parameter λ and length of the message, $|m|$.
An FE scheme that satisfies the above efficiency property is termed as compact FE. It was shown by [**?**,**?**] that iO is implied by (sub-exponentially hard) compact FE. However, we don't place any sub exponential hardness requirement on compact FE in our work.

Remark 1. We note that the definitions of FE for circuits commonly used in the literature do not have the above efficiency property.

Security. The security definition is modeled as a game between the challenger and the adversary as before.

Definition 2. *A public-key functional encryption scheme* $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *over a function space* $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda\in\mathbb{N}}$ *and a message space* $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$ *is an* ***adaptively-secure public-key functional encryption scheme*** *if for any PPT adversary* $\mathcal{A}$ *there exists a negligible*

function $\mu(\lambda)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, the advantage of $\mathcal{A}$ is defined to be

$$\mathsf{Adv}^{\mathsf{FE}}_{\mathcal{A}} = \left|\mathsf{Prob}[\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, 0) = 1] - \mathsf{Prob}[\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda),$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$ the experiment $\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(1^\lambda, b)$, modeled as a game between the challenger and the adversary $\mathcal{A}$, is defined as follows:

1. *The challenger first executes $\mathsf{Setup}(1^\lambda)$ to obtain $(\mathsf{PK}, \mathsf{MSK})$. It then sends PK to the adversary.*
2. ***Query Phase I**: The adversary submits a function query f to the challenger. The challenger sends back sk_f to the adversary, where sk_f is the output of $\mathsf{KeyGen}(\mathsf{MSK}, f)$.*
3. ***Challenge Phase**: The adversary submits a message-pair (m_0, m_1) to the challenger. The challenger checks whether $f(m_0) = f(m_1)$ for all function queries f made so far. If this is not the case, the challenger aborts. Otherwise, the challenger sends back $\mathsf{CT} = \mathsf{Enc}(\mathsf{MSK}, m_b)$.*
4. ***Query Phase II**: The adversary submits a function query f to the challenger. The challenger generates sk_f, where sk_f is the output of $\mathsf{KeyGen}(\mathsf{MSK}, f)$. It sends sk_f to the adversary only if $f(m_0) = f(m_1)$, otherwise it aborts.*
5. *The output of the experiment is b', where b' is the output of $\mathcal{A}$.*

We define the FE scheme to be selectively secure if the adversary has to declare the challenge message pair even before it receives the public parameters.

Function-private Private Key FE We now give an analogous definition of FE for circuits in the private-key setting. In particular, we focus on the private-key FE that is function-private.

A function-private private-key functional encryption (FE) scheme PrivFE, defined for a class of functions $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda\in\mathbb{N}}$ and message space $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$, is represented by four PPT algorithms, namely ($\mathsf{PrivFE.Setup}$, $\mathsf{PrivFE.KeyGen}$, $\mathsf{PrivFE.Enc}$, $\mathsf{PrivFE.Dec}$). The input length of any $f \in \mathcal{F}_\lambda$ is the same as the length of any $m \in \mathcal{M}_\lambda$.

We give the description of the four algorithms below.

- $\mathsf{PrivFE.Setup}(1^\lambda)$: It takes as input a security parameter λ in unary and outputs a secret key $\mathsf{PrivFE.MSK}$.
- $\mathsf{PrivFE.KeyGen}(\mathsf{PrivFE.MSK}, f \in \mathcal{F}_\lambda)$: It takes as input a secret key $\mathsf{PrivFE.MSK}$, a function $f \in \mathcal{F}_\lambda$ and outputs a functional key $\mathsf{PrivFE}.sk_f$.
- $\mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m \in \mathcal{M}_\lambda)$: It takes as input a secret key $\mathsf{PrivFE.MSK}$, a message $m \in \mathcal{M}_\lambda$ and outputs an encryption of m.
- $\mathsf{PrivFE.Dec}(\mathsf{PrivFE}.sk_f, \mathsf{CT})$: It takes as input a functional key $\mathsf{PrivFE}.sk_f$, a ciphertext CT and outputs $\hat{m}$.

We require the above function-private private key FE scheme to satisfy the correctness, efficiency and the function privacy properties of the above FE scheme.

Correctness. The correctness notion of a function-private private-key FE scheme dictates that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, for every message $m \in \mathcal{M}_\lambda$, and for every function $f \in \mathcal{F}_\lambda$ it holds that $\Pr[f(m) \leftarrow \mathsf{PrivFE.Dec}(\mathsf{PrivFE.KeyGen}(\mathsf{PrivFE.MSK}, f), \mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m))] \geq 1 - \mathsf{negl}(\lambda)$, where $\mathsf{PrivFE.MSK} \leftarrow \mathsf{PrivFE.Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Efficiency. At a high level, the efficiency property says that the setup algorithm and the encryption algorithm is independent of the size of the circuits for which functional keys are produced. More formally, the running time of $\mathsf{PrivFE.Setup}(1^\lambda)$ is just a polynomial in the security parameter λ, and $\mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m)$ is a polynomial in only the security parameter λ and length of the message, $|m|$.

Function Privacy. We now recall the definition of function privacy in private key FE as defined by Brakerski, and Segev [**?**].
Note that the function privacy property below subsumes the usual notion of security (when only one function is submitted).

Definition 3. *A private-key functional encryption scheme* $\mathsf{PrivFE} = (\mathsf{PrivFE.Setup}, \mathsf{PrivFE.KeyGen}, \mathsf{PrivFE.Enc}, \mathsf{PrivFE.Dec})$ *over a function space* $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda \in \mathbb{N}}$ *and a message space* $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda \in \mathbb{N}}$ *is a* ***function-private adaptively-secure private-key FE scheme*** *if for any PPT adversary* $\mathcal{A}$ *there exists a negligible function* $\mu(\lambda)$ *such that for all sufficiently large* $\lambda \in \mathbb{N}$*, the advantage of* $\mathcal{A}$ *is defined to be*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PrivFE}} = \left|\mathsf{Prob}[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PrivFE}}(1^\lambda, 0) = 1] - \mathsf{Prob}[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PrivFE}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda),$$

where for each $b \in \{0, 1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PrivFE}}(1^\lambda, b)$*, modeled as a game between the challenger and the adversary* $\mathcal{A}$*, is defined as follows:*

1. *The challenger first executes* $\mathsf{PrivFE.MSK} \leftarrow \mathsf{PrivFE.Setup}(1^\lambda)$*. The adversary then makes the following message queries and function queries in no particular order.*
   - ***Message queries****: The adversary submits a message-pair* (m_0, m_1) *to the challenger. In return, the challenger sends back* $\mathsf{CT} = \mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m_b)$*.*
   - ***Function queries****: The adversary then makes functional key queries. For every function-pair query* (f_0, f_1)*, the challenger sends* $\mathsf{PrivFE}.sk_{f_b}$ *to the adversary, where* $\mathsf{PrivFE}.sk_{f_b}$ *is the output of* $\mathsf{PrivFE.KeyGen}(\mathsf{PrivFE.MSK}, f_b)$ *only if* $f_0(m_0) = f_1(m_1)$*, for all message-pair queries* (m_0, m_1)*. Otherwise, it aborts.*
2. *The output of the experiment is* b'*, where* b' *is the output of* $\mathcal{A}$*.*

We define a function-private private key FE to be selectively secure if the adversary has to declare all the challenge message pairs at the beginning of the security game.

Remark 2. We note that we can define a private-key FE scheme without the function privacy property, analogous to the public-key FE.

Single-key setting. A single-key function-private functional encryption scheme (in the private-key setting) is a functional encryption scheme, where the adversary in the security game (either selective or adaptive) is allowed to query for only one function. There are several known constructions [**?**,**?**,**?**] but none of them satisfy the efficiency property of our FE definition – in particular, the size of the ciphertexts in these constructions grow with the circuit size (for which functional keys are computed). We later describe how to obtain a single-key scheme that indeed satisfies the efficiency property.

3 Adaptive 1-Key 1-Ciphertext FE for TMs

One of the main tools in our constructions is a single-key single-ciphertext FE for TMs in the private key setting. In the security game, the adversary only gets to make a single message and function query. Since we are interested in adaptive security, the message and the function query can be made in any order. In the language of randomized encodings (RE), this primitive is nothing but an adaptively secure *succinct* decomposable RE. The formal definition of single-ciphertext single-key FE for TMs is provided in the full version [**?**].
In the adaptive security game of single-ciphertext single-key FE, the adversary can only make a single function query and a single challenge message query. We define this notion for the case when the functions are represented by Turing machines.

As before, we can define a single-ciphertext single-key private-key FE to be *selectively-secure* if the adversary has to declare the challenge message pair even before he submits the function query.

We now proceed to build this tool based on iO and one-way functions. Towards this end, we first consider the notion of private key multi-ary functional encryption (FE) [**?**] for TMs. Multi-ary FE is a generalization of FE where the functions can take more than one input. We are interested in the restricted setting when the adversary only makes a single function and message query. Moreover, we restrict ourselves to the 2-ary setting, i.e., the arity of the functions is 2. We refer to this notion as 2-ary FE for TMs. We describe this notion formally in Section 3.1.
Prior to this work, we knew how to construct this only based on (public coins) differing inputs obfuscation. Later we show how to construct this primitive assuming just iO for circuits and one-way functions.

3.1 Semi-Adaptive 2-ary FE for TMs: 1-Key 1-Ciphertext Setting

The formal description of the 2-ary FE for TMs is given below. A 2-ary FE for a class of Turing machines $\mathcal{F}$ consists of four PPT algorithms, $\mathsf{2FE} = (\mathsf{2FE.Setup}, \mathsf{2FE.Enc}, \mathsf{2FE.KeyGen}, \mathsf{2FE.Dec})$, as described below.

- $\mathsf{2FE.Setup}(1^\lambda)$: On input the security parameter λ, the algorithm $\mathsf{2FE.Setup}$ outputs a master secret key $\mathsf{2FE.MSK}$.

- $\mathsf{2FE.KeyGen}(\mathsf{2FE.MSK}, M)$: On input the master secret key $\mathsf{2FE.MSK}$ and Turing machine $M \in \mathcal{F}$, it outputs the key $\mathsf{2FE.sk}_M$.
- $\mathsf{2FE.Enc}(\mathsf{2FE.MSK}, x, b)$: On input the master secret key $\mathsf{2FE.MSK}$, message $x \in \{0,1\}^*$ and position $b \in \{0,1\}$, it outputs $\mathsf{2FE.CT}_x$.

  Remark 3. The bit b essentially indicates the position with respect to which the message needs to be encrypted. For convenience sake, we refer to the first position as the 0^{th} position and the second position as the 1^{st} position.
- $\mathsf{2FE.Dec}(\mathsf{2FE.sk}_M, \mathsf{2FE.CT}_x, \mathsf{2FE.CT}_y)$: On input the functional key $\mathsf{2FE.sk}_M$ and ciphertexts $\mathsf{2FE.CT}_x$ and $\mathsf{2FE.CT}_y$, it outputs the value z.

For the above notion to be interesting, a 2-ary FE for TMs scheme is required to satisfy the following correctness, efficiency and security properties.

Correctness: This property ensures that the output of $\mathsf{2FE.Dec}(\mathsf{2FE.sk}_M, \mathsf{2FE.CT}_x, \mathsf{2FE.CT}_y)$ is always $M(x,y)$ where (i) $\mathsf{2FE.MSK} \leftarrow \mathsf{2FE.Setup}(1^\lambda)$, (ii) $\mathsf{2FE.sk}_M \leftarrow \mathsf{2FE.KeyGen}(\mathsf{2FE.MSK}, M)$, (iii) $\mathsf{2FE.CT}_x \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, x, 0)$ and (iv) $\mathsf{2FE.CT}_y \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, y, 1)$.

Efficiency: This property says that the size of the ciphertexts (resp., functional key) depend solely on the size of the message (resp., machine) and the security parameter. That is, the complexity of $\mathsf{2FE.Enc}(\mathsf{2FE.MSK}, x, b)$ is a polynomial in $(\lambda, |x|)$ and the complexity of $\mathsf{2FE.KeyGen}(\mathsf{2FE.MSK}, M)$ is a polynomial in $(\lambda, |M|)$. Furthermore, we require that the complexity of $\mathsf{2FE.Dec}(\mathsf{2FE.sk}_M, \mathsf{2FE.CT}_x, \mathsf{2FE.CT}_y)$ is just a polynomial in $(\lambda, |x|, |y|, |M|, t)$, where t is the time taken by M to execute on the input (x,y).

Semi-Adaptive Security: The security guarantee states that the adversary cannot distinguish joint ciphertexts of (x_0, y_0) from the joint ciphertexts of (x_1, y_1) given the functional key of M, as long as $M(x_0, y_0) = M(x_1, y_1)$. Note that we adopt the convention that the Turing machine also outputs its running time and thus this alone ensures that the execution time of $M(x_0, y_0)$ is the same as the execution time of $M(x_1, y_1)$. Depending on the order of the message and the Turing machine queries the adversary can make, there are many ways to model the security of a 2-ary FE scheme. We adopt the notion where the adversary can make the message queries corresponding to 0^{th} and 1^{st} position in an adaptive manner but the TM query should be made *only after both the message queries*. We term this notion *semi-adaptive* security.
Suppose $\mathcal{A}$ be any PPT adversary. We define an experiment $\mathsf{Expt}_{\mathcal{A}^{\mathsf{SemiAd}}}$ below.

$\underline{\mathsf{Expt}^{\mathsf{SemiAd}}_{\mathcal{A}}(1^\lambda)}$:

1. The challenger first executes $\mathsf{2FE.Setup}(1^\lambda)$ to obtain $\mathsf{2FE.MSK}$. It then chooses a bit b at random.
2. The following two bullets are executed in an arbitrary order (depending on the choice of the adversary).

- The adversary submits the message query (x_0, x_1), corresponding to 0^{th} position, to the challenger. The challenger responds with $\mathsf{2FE.CT}_x \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, x_0, 0)$ if $b = 0$ else it responds with $\mathsf{2FE.CT}_x \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, x_1, 0)$.
- The adversary submits the message query (y_0, y_1), corresponding to 1^{st} position, to the challenger. The challenger responds with $\mathsf{2FE.CT}_y \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, y_0, 1)$ if $b = 0$ else it responds with $\mathsf{2FE.CT}_y \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, y_1, 1)$.

3. After both the message queries, the adversary then submits a Turing machine M to the challenger. The challenger aborts if either (i) $M(x_0, y_0) \neq M(x_1, y_1)$ or (ii) $|x_0| \neq |x_1|$ or (iii) $|y_0| \neq |y_1|$. If it has not aborted, it executes $\mathsf{2FE.sk}_M \leftarrow \mathsf{2FE.KeyGen}(\mathsf{2FE.MSK}, M)$. It then sends $\mathsf{2FE.sk}_M$ to the adversary.
4. The adversary outputs b'.

The experiment outputs 1 if $b = b'$, otherwise it outputs 0.
We now define the semi-adaptive security notion.

Definition 4. *A 2-ary FE scheme is semi-adaptive secure if for any PPT adversary $\mathcal{A}$, we have that the probability that the output of the experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SemiAd}}$ is 1 is at most $1/2 + \mathsf{negl}(\lambda)$, for any negligible function negl.*

3.2 Adaptive FE from Semi-Adaptive 2-ary FE for TMs

We now show how to achieve *adaptively secure* single-ciphertext single-key FE starting from a *semi-adaptively secure* 2-ary FE for TMs. Recall that in the semi-adaptive security game of 2-ary FE, the key query can be made only after the message queries but however, the message queries corresponding to the first and the second position can be made in an adaptive manner. This leads to the main idea behind our construction – symmetrization of the input and the TM. That is, the adaptive FE functional key of a machine M is the 2-ary FE encryption of M w.r.t the 1^{st} position and the adaptive FE encryption of a message m is essentially the 2-ary FE encryption of m w.r.t the 0^{th} position. This takes care of the adaptivity issue. To facilitate the execution of M on m, a 2-ary FE key of a universal TM (UTM) is also provided. The question is whether we include the 2-ary FE key of UTM in the ciphertext or the functional key. This is crucial because the UTM key can only be provided by the challenger after seeing the queries corresponding to both the 0^{th} and 1^{st} position. To solve this issue, we additively secret share the UTM key across both the ciphertext and the functional key. This gives the challenger leeway to provide a random string as part of the response to the first query and by providing the appropriate secret share in the second response it can reveal the UTM key – at this point the challenger has seen both m and M. The formal scheme is described next.
Consider a 2-ary FE for TMs, denoted by $\mathsf{2FE} = (\mathsf{2FE.Setup}, \mathsf{2FE.KeyGen}, \mathsf{2FE.Enc}, \mathsf{2FE.Dec})$, for a class of Turing machines $\mathcal{F}$. We construct a single-ciphertext single-key FE, $\mathsf{OneCTKey}$, for the same class $\mathcal{F}$.
Denote by $\mathsf{UTM} = \mathsf{UTM}_\lambda$, the universal Turing machine, that takes as input a Turing machine M, message m and outputs $M(m)$ if it halts

within 2^λ steps else it outputs $\perp$. Further, we denote by ℓ_{UTM} to be the length of the output of a $\mathsf{2FE}$ key of UTM.

$\underline{\mathsf{OneCTKey.Setup}(1^\lambda)}$: On input the security parameter λ, it first executes $\mathsf{2FE.Setup}(1^\lambda)$ to obtain the master secret key $\mathsf{2FE.MSK}$. It also picks a random string R in $\{0,1\}^{\ell_{\mathsf{UTM}}}$. It outputs the secret key $\mathsf{OneCTKey.MSK} = (\mathsf{2FE.MSK}, R)$ as the master secret key.

$\underline{\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, M \in \mathcal{F})}$: On input the master secret key $\mathsf{OneCTKey.MSK} = (\mathsf{2FE.MSK}, R)$, and a Turing machine $M \in \mathcal{F}$, it executes 2-ary FE encryption of M w.r.t 0^{th} position, $\mathsf{2FE.Enc}(\mathsf{2FE.MSK}, M, 0)$, to obtain $\mathsf{2FE.CT}_M$. It then computes a 2-ary FE key of UTM by generating $\mathsf{2FE.sk}_{\mathsf{UTM}} \leftarrow \mathsf{2FE.KeyGen}(\mathsf{2FE.MSK}, \mathsf{UTM}_\lambda)$. Finally, it outputs the functional key $\mathsf{OneCTKey}.sk_M = (\mathsf{2FE.CT}_M, \mathsf{2FE.sk}_{\mathsf{UTM}} \oplus R)$.

$\underline{\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m)}$: On input the master secret key $\mathsf{OneCTKey.MSK} = (\mathsf{2FE.MSK}, R)$, and message m, it generates a 2-ary FE encryption of m by executing $\mathsf{2FE.CT}_m \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, m, 1)$. It outputs the ciphertext $\mathsf{OneCTKey.CT} = (\mathsf{2FE.CT}_m, R)$.

$\underline{\mathsf{OneCTKey.Dec}(\mathsf{OneCTKey}.sk_M, \mathsf{OneCTKey.CT})}$: On input the functional key $\mathsf{OneCTKey}.sk_M = (\mathsf{2FE.CT}_M, S)$ and ciphertext $\mathsf{OneCTKey.CT} = (\mathsf{2FE.CT}_m, R)$. It computes $S \oplus R$ to obtain $\mathsf{2FE.sk}_{\mathsf{UTM}}$. It then executes $\mathsf{2FE.Dec}(\mathsf{2FE.sk}_{\mathsf{UTM}}, \mathsf{2FE.CT}_M, \mathsf{2FE.CT}_m)$ to obtain z. Finally, it outputs z.

We prove the following theorem. The proof of the theorem is available in the full version [**?**].

Theorem 2. *The scheme* $\mathsf{OneCTKey}$ *satisfies correctness, efficiency and adaptive security properties.*

3.3 Constructing Semi-Adaptive 2-ary FE for TMs: Overview

Lets begin with the following simple idea: the 2-ary FE encryption of x w.r.t 0^{th} position will just be a standard public key encryption of x_0. Since this encryption should not be malleable, we provide an authentication of the ciphertext. Similarly, the 2-ary FE encryption of y w.r.t 1^{st} position is also a public key encryption of y along with its authentication. The functional key of M is an obfuscated program that takes as input an encrypted tape symbol; decrypts it; executes the next message function and then outputs an encryption of the new symbol. The evaluation is performed by executing next message function one step at a time while updating the storage tape which is initialized to the encryptions of x and y along with their respective authentications.
This however suffers from consistency issues. An adversary could re-use encrypted storage tape values of the current tape in the future steps. It would seem that using signatures to bind the time step to the tape symbol

should solve this problem. In fact, if we had virtual black box obfuscation this idea would work. However, we are stuck with indistinguishability obfuscation and it is not clear how to make this work – signatures in general aren't compatible with iO because signatures guarantee computational soundness whereas iO demands information theoretic soundness. Looking back at the literature, we notice that Koppula-Lewko-Waters had to deal with similar issues in their recent work on randomized encodings (RE)[4] for TMs [**?**]. The template of their construction comprises of two components as described below. The actual construction of KLW has more intricate details involved from what is presented below but to keep the discussion at an intuitive level, we choose to describe it this way.
Let M and x be the input to the encoding procedure.

- **Storage tree**: Encrypt x using a public key encryption scheme. Initialize the work tape with this ciphertext. Compute a storage tree on this ciphertext. The root of the storage tree along with the current time step (which is initially 0) is then signed using a signature scheme. This signature serves as an authentication of the work tape and the current time step.
- **Obfuscated next message program**: The obfuscated program takes as input an encrypted tape symbol (leaf node), its path to the root of the storage tree and the signature on the root. It performs few checks to test whether the encrypted tape symbol is valid. It then decrypts the encrypted tape symbol, computes the next message function of the TM M and then re-encrypts the output tape symbol. Finally, it computes the new root of the storage tree (this can be done by just having the appropriate path from the new tape symbol leading up to the root) and signs it.

There are two main hurdles in using the above template for our construction of 2-ary FE for TMs: (i) the TM only takes a single input in the above template whereas in our setting the TM takes two inputs. Moreover, we require that the TM and the inputs are encoded separately and, (ii) the security notion considered by KLW is *weak-selective* – the adversary is required to declare both the TM and the input at the beginning of the game. On the other hand the security notion we consider is stronger. Because of these two main reasons, we employ new techniques to achieve our construction.

Ciphertext combiner mechanism. As remarked earlier, we require that the TM and the inputs are encoded separately. We exploit the fact that inherently KLW has two components – storage tree and obfuscated next message program – that depend upon the input and the TM separately. But note that we have two inputs and so we need to further split the storage tree component. The tree structure automatically allows for such a decomposition. We compute a storage tree on the (encrypted) 0^{th} position input and another tree on the (encrypted) 1^{st} position input. We can then combine the roots of both the trees, during the decryption phase, to

[4] A randomized encoding of a machine M and input x is an encoding of $M(x)$ that takes much less time to compute than $M(x)$. Furthermore, the encoding should only reveal $M(x)$ and nothing more.

obtain a new root. But the root of the new tree needs to be authenticated and this operation needs to be public. We could provide the decryptor the signing key but then we end up sacrificing security!
To overcome this problem, we provide a combiner program, as part of one of the ciphertexts, that takes as input two nodes in the tree and outputs a new node along with a signature. This signature is signed using a signing key which is part of the combiner program. Of course the combiner program needs to be obfuscated to hide the signing key. As we will see later in the actual construction, we require "iO-compatible" signatures a.k.a splittable signatures scheme of KLW to make this idea work.
While using combiner seems to solve the problem, the next question is in which ciphertext do we include the combiner? We will see next that this becomes crucial for our proof of security.

Ensuring semi-adaptivity. Suppose we decide to include the combiner as part of the 0^{th} ciphertext. In line with the techniques used in proving the security using iO, we require that in the proof of security we hardwire the resulting (combined) root node in the combiner. But this is not possible if the 0^{th} position challenge message is requested before the 1^{st} position challenge message. The same problem occurs if we include the combiner as part of the 1^{st} position ciphertext – the adversary can now query for the 1^{st} position challenge ciphertext first and then query the 0^{th} position challenge message.
This conundrum can be tackled by using deniable encryption. We can compute a deniable encryption of combiner in one ciphertext and in the other ciphertext we open the deniable ciphertext. This gives us the flexibility to open the ciphertext to whatever message we want depending on the adversary's queries. While this solves the problem, we can replace deniable encryption with a much simpler tool – one-time pad! We compute a one-time pad of the combiner with randomness R in one ciphertext and the other ciphertext contains just R. This solves our problem just like the case of deniable encryption.

We present a high level and a simplified description of the 2-ary FE scheme below. The formal description is more involved and is presented in full version [**?**] where we present the construction in a modular fashion by first describing an intermediate primitive that we call 3-stage KLW.

1. **Setup**: Generate a master signing key-verification key pair (SK, VK). Also generate two auxiliary signature key-verification key pairs (SK_x, VK_x) and (SK_y, VK_y). Generate the public parameters PP of the storage tree. Compute a random string R of appropriate length. The public key is PP while the master secret key is $(SK_x, SK_y, VK_x, VK_y, SK, VK, R)$.
2. **Key generation of** M: Generate an obfuscated next message program of M whose functionality is as in the above high level description. The pair (SK, VK) is hardwired inside the obfuscated program.
3. **Encryption of** x **w.r.t** 0^{th} **position**: Compute a storage tree on x. Sign the root of the tree rt_x using SK_x to obtain σ_x. Compute the obfuscated combiner program $S = \mathsf{Comb} \oplus R$ whose description is as given above. Output $(\mathsf{rt}_x, \sigma_x, S)$.

4. **Encryption of y w.r.t 1^{st} position**: Compute a storage tree on y. Sign the root of the tree rt_y using SK_y to obtain σ_y. Output $(\mathsf{rt}_y, \sigma_y, R)$.
5. **Decryption**: First, compute $S \oplus R$ to recover Comb. Then execute Comb on inputs $((\mathsf{rt}_x, \sigma_x), (\mathsf{rt}_y, \sigma_y))$ to obtain the joint root rt accompanied by the signature σ computed using SK. Once this is done, using the joint tree and obfuscated next message program of M, execute the decode procedure of KLW to recover the answer.

4 Adaptive FE for TMs

We show how to obtain an adaptively secure public key functional encryption scheme for Turing machines. To achieve this, we use a public key FE scheme for circuits, single-key FE scheme for circuits and single-key single-ciphertext FE for Turing machines.

Tools. We use the following tools to achieve the transformation.

- (Compact) Public key FE scheme for circuits, denoted by $\mathsf{PubFE} = (\mathsf{PubFE.Setup}, \mathsf{PubFE.KeyGen}, \mathsf{PubFE.Enc}, \mathsf{PubFE.Dec})$. It suffices for us that PubFE is selectively secure.
- (Compact) Function-private Single-key FE scheme for circuits, denoted by $\mathsf{OneKey} = (\mathsf{OneKey.Setup}, \mathsf{OneKey.KeyGen}, \mathsf{OneKey.Enc}, \mathsf{OneKey.Dec})$. It suffices for us that OneKey is selectively secure.
- Single-key single-ciphertext FE scheme for Turing machines, denoted by $\mathsf{OneCTKey} = (\mathsf{OneCTKey.Setup}, \mathsf{OneCTKey.KeyGen}, \mathsf{OneCTKey.Enc}, \mathsf{OneCTKey.Dec})$. We require that $\mathsf{OneCTKey}$ is adaptively secure.
- Psuedorandom function family, F.
- Symmetric encryption scheme with pseudorandom ciphertexts, denoted by $\mathsf{Sym} = (\mathsf{Sym.Setup}, \mathsf{Sym.Enc}, \mathsf{Sym.Dec})$.

Instantiations of the tools. We gave an construction of single-key single-ciphertext FE for Turing machines satisfying adaptive security in Section 3. We can instantiate the compact public-key FE scheme using the construction of [**?**,**?**] (here, we refer to the post-challenge FE construction of [**?**]). This construction can be based on indistinguishability obfuscation and other standard assumptions. Lastly, we can instantiate a function-private single key FE by, first, applying the function-privacy transformation by Brakerski-Segev [**?**] on the public-key FE [5]. The resulting FE is a private-key FE which is also function-private. And, a function-private single-key FE in the private key setting is a special case of function-private private key FE. Note that this instantiation can be based off the same assumptions as public-key FE (this is because, [**?**] does not add any additional assumptions).

[5] The function-privacy transformation was defined for private key FE but a public key FE can be transformed into a private key FE in a straightforward way.

Intuition. We view our construction as a transformation from adaptively secure 1-CT 1-key FE scheme into one that can handle unbounded collusions. Even though in general we don't know any way of achieving this, we show that by leveraging additional tools we can attain this goal. These additional tools, as mentioned above, are multi-key FE schemes that are only selective secure.
The key idea is as follows: we give a mechanism to generate a *unique key* corresponding to a pair of ciphertext (of m) and functional key (of f) in the resulting adaptive multi-key FE scheme. This unique key would correspond to the master secret key of the adaptive 1-CT 1-Key FE scheme. At this point, we can generate functional keys of f and ciphertext of m w.r.t this unique key. Implementing this mechanism using iO, in the context of FE for circuits, was introduced by Waters [**?**]. We show how to implement the same, in the more general context of FE for TMs, but using just a multi-key FE. We highlight that in general we don't know how to replace the use of iO with multi-key FE since FE does not offer function hiding.
At the high level, the construction proceeds as follows. A ciphertext of m "communicates" a PRF key K to a functional key of f. This communication is enabled by a multi-key FE scheme. The functional key using K and hardwired values, derives the master secret key $\mathsf{OneCTKey.MSK}$ of a 1-CT 1-Key FE scheme. If then computes a functional key of f w.r.t $\mathsf{OneCTKey.MSK}$. But the ciphertext of m does not contain an encryption w.r.t $\mathsf{OneCTKey.MSK}$! And so this key has to be "communicated" from functional key back to the ciphertext. To do this, we will use another instantiation of selectively secure FE scheme. Here, we note that it suffices to consider just a single-key scheme and that too in the private key setting. Once we have this instantiation, the functional key can now generate a single-key FE encryption of $\mathsf{OneCTKey.MSK}$. The single-key FE functional key, which will now be part of the ciphertext, will take as input encryption of $\mathsf{OneCTKey.MSK}$ and outputs an encryption of m w.r.t $\mathsf{OneCTKey.MSK}$. Finally, we can just run the decryption algorithm of $\mathsf{OneCTKey}$ to obtain the answer. We illustrate a simple example, when a single ciphertext and functional key is released, in Figure 1.
Our construction has more details that we present below.

Construction. We now describe the construction. We denote the FE for TMs scheme, that we construct, to be $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

$\underline{\mathsf{Setup}(1^\lambda)}$: Execute $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$.
Output the secret key-public key pair $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.

$\mathsf{KeyGen}(\mathsf{MSK} = \mathsf{PubFE.MSK}, f)$: Draw C_E at random[6]. Denote τ to be $\overline{(\tau_0||\tau_1||\tau_2||\tau_3)}$, where τ_i for $i \in \{0,1,2,3\}$ is picked at random. Execute

[6] The length of C_E is determined as follows. Denote by $|f|$, the size of the Turing machine representing f. Denote by ℓ_{OneCTKey}, the length of the ciphertext obtained by encrypting a message of length $|f|$, using $\mathsf{OneCTKey.Enc}$. Denote by ℓ_{OneKey}, the length of the ciphertext obtained by encrypting a message of length $\lambda + 2$, using $\mathsf{OneKey.Enc}$. Further, denote by ℓ_{Sym} to be the length of the ciphertext obtained by

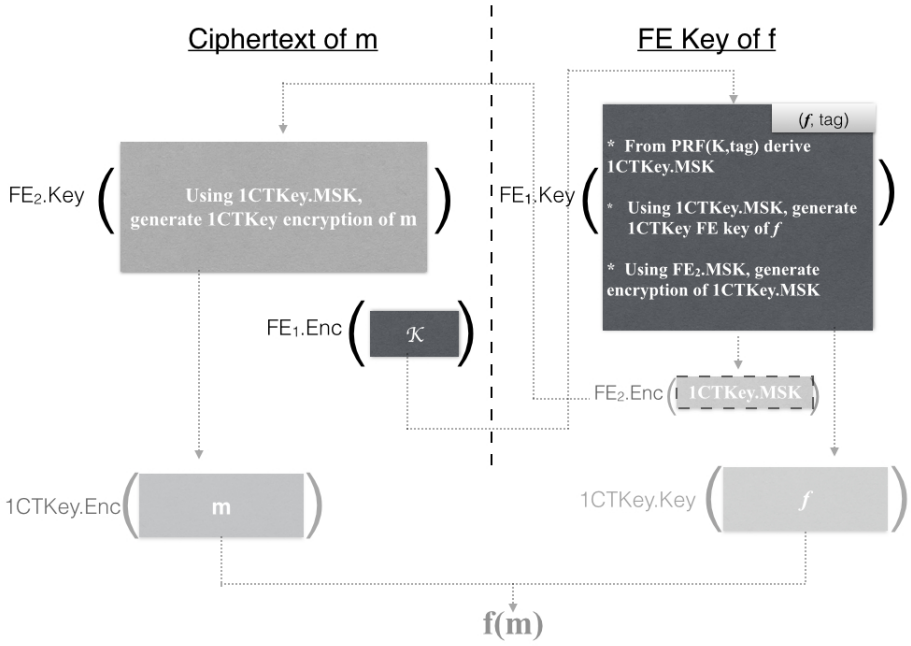

Fig. 1 The ciphertext of m has two components – the first component is a single-key FE (denoted by FE_2) functional key and the second component is a multi-key FE (denoted by FE_1) encryption of a PRF key K. The function key of f is just a FE_1 functional key of the program described in the figure. The arrows indicate the flow of execution of decryption of the ciphertext of m using the functional key of f.

$\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f, C_E, \tau])$, where $G[f, C_E, \tau]$ is described in Figure 2, to obtain $\mathsf{PubFE}.sk_G$. Output $sk_f = \mathsf{PubFE}.sk_G$.

$\underline{\mathsf{Enc}(\mathsf{PK} = \mathsf{PubFE.PK}, m)}$:

- Draw a PRF key K at random from $\{0,1\}^\lambda$.
- Execute $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}$.
- Execute $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}, H[m])$ to obtain $\mathsf{OneKey}.sk_H$, where $H[m]$ is defined in Figure 3.
- Execute $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\mathsf{OneKey.MSK}, \mathsf{K}, \perp, 0))$ to obtain $\mathsf{PubFE.CT}$.

Finally, output $\mathsf{CT} = (\mathsf{OneKey}.sk_H, \mathsf{PubFE.CT})$.

$\underline{\mathsf{Dec}(sk_f = sk_G, \mathsf{CT} = (\mathsf{OneKey}.sk_H, \mathsf{PubFE.CT}))}$:

- Execute $\mathsf{PubFE.Dec}(\mathsf{PubFE}.sk_G, \mathsf{PubFE.CT})$ to obtain $(\mathsf{OneCTKey}.sk_f, \mathsf{OneKey.CT})$.

encrypting a message of length $\ell_{\mathsf{OneCTKey}} + \ell_{\mathsf{OneKey}}$, using $\mathsf{Sym.Enc}$. We set the length of C_E to be ℓ_{Sym}.

$\underline{G[f, C_E, \tau](\mathsf{OneKey.MSK}, \mathsf{K}, \mathsf{Sym}.k, \beta)}$

1. Parse τ as $(\tau_0||\tau_1||\tau_2||\tau_3)$
2. If $\beta = 0$ then
   - $R_i \leftarrow \mathsf{PRF}(\mathsf{K}, \tau_i)$, for $i \in \{0, 1, 2, 3\}$
   - $\mathsf{OneCTKey.MSK} \leftarrow \mathsf{OneCTKey.Setup}(1^\lambda; R_0)$
   - $\mathsf{OneCTKey}.sk_f \leftarrow \mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R_1)$
   - $\mathsf{OneKey.CT} \leftarrow \mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}, (\mathsf{OneCTKey.MSK}, R_2, 0); R_3)$
   - Output $(\mathsf{OneCTKey}.sk_f, \mathsf{OneKey.CT})$
3. Else,
   - $(\mathsf{OneCTKey}.sk_f, \mathsf{OneKey.CT}) \leftarrow \mathsf{Sym.Dec}(\mathsf{Sym}.k, C_E)$
   - Output $(\mathsf{OneCTKey}.sk_f, \mathsf{OneKey.CT})$

Fig. 2

$\underline{H[m](\mathsf{OneCTKey.MSK}, R, \alpha)}$

1. If $\alpha = 0$ then
   - $\mathsf{OneCTKey.CT} \leftarrow \mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m; R)$
   - Output $\mathsf{OneCTKey.CT}$
2. Else, output $\perp$.

Fig. 3

- Execute $\mathsf{OneKey.Dec}(\mathsf{OneKey}.sk_H, \mathsf{OneKey.CT})$ to obtain $\mathsf{OneCTKey.CT}$.
- Execute $\mathsf{OneCTKey.Dec}(\mathsf{OneCTKey}.sk_f, \mathsf{OneCTKey.CT})$ to obtain $\hat{m}$.

Output $\hat{m}$.

We prove the following theorem that establishes the proof of security of the above scheme.

Theorem 3. *Assuming the selective security of* PubFE, OneKey, *adaptive security of* OneCTKey, *security of* F, Sym, *we have that the scheme* FE *is adaptively secure.*

Since the proof is involved, we choose to first present the proof of selective security of FE. We then point out the (minor) changes that need to be made to prove the adaptive security of FE. We give a sketch of the proof of the above scheme in Section 5 and the formal proof is provided in the full version [**?**]. We also present the proof of correctness and efficiency in the full version.

5 Proof of Theorem 3: Overview

To explain the proof intuition, we restrict ourselves to the setting when the adversary makes only a single message and key query.
In the first hybrid, the challenger uses a bit b picked at random, to generate the challenge ciphertext as in the (selective) security notion. By using the security of many primitives (listed in the theorem statement),

we then move to a hybrid where the bit b is information-theoretically hidden from the adversary. At this point, the probability that the adversary guesses the bit b is $1/2$. And thus the probability that the adversary guesses b correctly in the first hybrid is at most $1/2 + \mathsf{negl}(\lambda)$.

Hybrid_0: This corresponds to the real experiment when the challenger uses the b^{th} message in the challenge message pair query to compute the challenge ciphertext, where the bit b is picked at random. The output of the hybrid is the same as the output of the adversary.

Hybrid_1: In this hybrid, the values corresponding to the challenge ciphertext are hardwired in the "C_E" component of all the functional keys.
That is, the challenger upon receiving a function query f, first samples a symmetric key $\mathsf{Sym}.k^*$. It generates an encryption of the message $(\mathsf{OneCTKey.MSK}, R_2, 0)$ with respect to the single-key FE scheme. Call this ciphertext, $\mathsf{OneKey.CT}$. It then samples a functional key of f with respect to the single-key single-ciphertext FE scheme. Call this functional key, $\mathsf{OneCTKey}.sk_f$. It is important to note here that, the (pseudo)randomness used in the generation of $\mathsf{OneKey.CT}$ and $\mathsf{OneCTKey}_f$ is as described in the scheme. Finally, it computes a symmetric encryption of $(\mathsf{OneKey.CT}, \mathsf{OneCTKey}.sk_f)$ using the key $\mathsf{Sym}.k$. The resulting ciphertext will be assigned to C_E and then the challenger proceeds as in the previous hybrid.
The indistinguishability of Hybrid_0 and Hybrid_1 follows from the security of symmetric encryption scheme.

Hybrid_2: In this hybrid, the mode is switched from $\beta = 0$ to $\beta = 1$.
Upon receiving a challenge message query (m_0, m_1), the challenger computes the challenge ciphertext as follows. Recall that there are two components in the ciphertext – namely, the single-key FE functional key and the public-key FE ciphertext. The challenger computes the single-key FE functional key as in the previous hybrid. However, it generates the public-key FE cipehertext to be an encryption of $(\bot, \bot, \mathsf{Sym}.k^*, 1)$ instead of $(\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \bot, 0)$, as in Hybrid_1. The rest of the hybrid is the same as the previous hybrid.
The indistinguishability of Hybrid_1 and Hybrid_2 follows from the security of public-key FE scheme. This is because the output of G (Figure 2) on input $(\bot, \bot, \mathsf{Sym}.k^*, 1)$ is nothing but the decryption of C_E. And by our choice of C_E, this is the same as the output of G on input $(\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \bot, 0)$.

Hybrid_3: The hardwired values in the "C_E" components of all the functional keys are now computed using randomness drawn from a uniform distribution. Recall that in the previous hybrid, the single-key ciphertext and the single-key single-ciphertext FE encrypted in C_E were computed using pseudorandom values.
The indistinguishability of Hybrid_2 and Hybrid_3 follows from the security of pseudorandom function family.

$\underline{\mathsf{Hybrid}_4}$: A branch encrypting message m_0 (the 0^{th} message in the challenge message query) is introduced in the function H.
The challenger upon receiving the challenge message query (m_0, m_1), first computes a single-key FE functional key of the function $H^*[m_0, m_b, v]$, as described in Figure 4. Here, b is the challenge bit, picked at random by the challenger. The program H^* is the same as H except that it contains an additional branch. The rest of the hybrid is the same as Hybrid_3.
The indistinguishability of Hybrid_3 and Hybrid_4 follows from the function-privacy property of single-key FE scheme. To see why, let us look at the messages that are encrypted under the single-key FE scheme (note that each encryption is part of the "C_E" component of some functional key). We observe that each message is of the form $(\mathsf{OneCTKey.MSK}, R, 0)$. From the descriptions of H and H^*, it follows that the output of H on $(\mathsf{OneCTKey.MSK}, R, 0)$ is the same as the output of H^* on $(\mathsf{OneCTKey.MSK}, R, 0)$.

$\underline{H^*[m, m', v](\mathsf{OneCTKey.MSK}, R, \alpha)}$

1. If $\alpha = 0$ then
   - $\mathsf{OneCTKey.CT} \leftarrow \mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m; R)$
   - Output $\mathsf{OneCTKey.CT}$
2. If $\alpha = 1$ then
   - $\mathsf{OneCTKey.CT} \leftarrow \mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m'; R)$
   - Output $\mathsf{OneCTKey.CT}$
3. Else, output v.

Fig. 4

$\underline{\mathsf{Hybrid}_5}$: We switch the mode of α from 0 to 1 in the OneKey ciphertexts output by all the functional keys.
The challenger, upon receiving a functional query f, first generates a single-key FE ciphertext to be an encryption of $(\mathsf{OneCTKey.MSK}, R, 1)$, where $\mathsf{OneCTKey.MSK}$ is as generated in the previous hybrids. The resulting ciphertext along with the single-key single-ciphertext FE functional key is then encrypted, using the symmetric key encryption, to obtain C_E. The rest of the functional key is then generated as previously.
The indistinguishability of Hybrid_4 and Hybrid_5 is more complex and involves more intermediate hybrids and thus we defer the explaination.

$\underline{\mathsf{Hybrid}_6}$: We change the $\alpha = 0$ branch in the function H to encrypt the message m_0 instead of m_b.
The challenger upon receiving a message query (m_0, m_1), first generates a single-key FE functional key of $H^*[m_0, m_0, v]$. It then generates the public key FE encryption as in previous hybrids. The rest of the hybrid is as in Hybrid_5.
The indistinguishability of Hybrid_5 and Hybrid_6 follows from the function privacy property of single-key FE scheme. To see why, we look at the

messages encrypted in the single-key FE ciphertexts. We first note all these ciphertexts are part of "C_E" component of some functional key. Further, each message is of the form $(\mathsf{OneCTKey.MSK}, R, 1)$. Thus, the output of $H^*[m_b, m_0, v]$ is the same as the output of $H^*[m_0, m_0, v]$.

Observe that the challenge bit b is no longer used. This combined with the indistinguishability of consecutive hybrids proves that the probability that $\mathcal{A}$ wins in Hybrid_1 is at most $1/2 + \mathsf{negl}(\lambda)$. This proves the security of FE.

6 Future Directions

The works of [**?**,**?**,**?**] show the equivalence of (sub-exponentially secure) FE and iO for the case of circuits. It would be interesting to explore the possibility of the equivalence of FE for Turing machines and iO for Turing machines (with no restriction on the input length). One direct consequence of a feasibility result in this direction is establishing the existence of iO for Turing machines based on iO for circuits. The current feasibility results on iO for Turing machines are based on knowledge assumptions.

Acknowledgements

We thank Brent Waters for collaboration at early stages of this project and several discussions. We also thank the anonymous reviewers of TCC 2016-A for their useful suggestions. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant CNS-1523467.

A Tools Used in [**?**]

We recall the key tools, namely, positional accumulators, iterators and splittable signatures, used in the work of Koppula et al. [**?**].
We now describe the syntax of the tools below. We refer the reader to [**?**] for the correctness and the security definitions.

A.1 Positional Accumulators

The notion of *positional accumulators* is defined below. A positional accumulator for message space Msg_λ consists of the following algorithms.

$\mathsf{SetupAcc}(1^\lambda, T) \to \mathsf{PP}_{\mathsf{Acc}}, w_0, store_0$ The setup algorithm takes as input a security parameter λ in unary and an integer T in binary representing the maximum number of values that can stored. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

$\mathsf{EnforceRead}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k), \mathrm{INDEX}^*) \rightarrow (\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0)$. The setup enforce read algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$, and an additional INDEX^* also between 0 and $T-1$. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

$\mathsf{EnforceWrite}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k)) \rightarrow \mathsf{PP}_{\mathsf{Acc}}, w_0, store_0$ The setup enforce write algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

$\mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \mathrm{INDEX}) \rightarrow m, \pi$ The prep-read algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, a storage value $store_{in}$, and an index between 0 and $T-1$. It outputs a symbol m (that can be ϵ) and a value π.

$\mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \mathrm{INDEX}) \rightarrow aux$ The prep-write algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, a storage value $store_{in}$, and an index between 0 and $T-1$. It outputs an auxiliary value aux.

$\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_{in}, m_{read}, \mathrm{INDEX}, \pi) \rightarrow \{True, False\}$ The verify-read algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an accumulator value w_{in}, a symbol, m_{read}, an index between 0 and $T-1$, and a value π. It outputs $True$ or $False$.

$\mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \mathrm{INDEX}, m) \rightarrow store_{out}$ The write-store algorithm takes in the public parameters, a storage value $store_{in}$, an index between 0 and $T-1$, and a symbol m. It outputs a storage value $store_{out}$.

$\mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, w_{in}, m_{write}, \mathrm{INDEX}, aux) \rightarrow w_{out}$ **or** $Reject$ The update algorithm takes in the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an accumulator value w_{in}, a symbol m_{write}, and index between 0 and $T-1$, and an auxiliary value aux. It outputs an accumulator value w_{out} or $Reject$.

A.2 Iterators

In this subsection, we now describe the notion of cryptographic iterators. As remarked earlier, iterators essentially consist of states that are updated on the basis of the messages received. We describe its syntax below.

Syntax Let ℓ be any polynomial. An iterator $\mathsf{PP}_{\mathsf{Itr}}$ with message space $\mathsf{Msg}_\lambda = \{0,1\}^{\ell(\lambda)}$ and state space $\mathsf{SplScheme}_\lambda$ consists of three algorithms - $\mathsf{SetupItr}$, $\mathsf{ItrEnforce}$ and $\mathsf{Iterate}$ defined below.

$\mathsf{SetupItr}(1^\lambda, T)$ The setup algorithm takes as input the security parameter λ (in unary), and an integer bound T (in binary) on the number of iterations. It outputs public parameters $\mathsf{PP}_{\mathsf{Itr}}$ and an initial state $v_0 \in \mathsf{SplScheme}_\lambda$.

$\mathsf{ItrEnforce}(1^\lambda, T, \boldsymbol{m} = (m_1, \ldots, m_k))$ The enforced setup algorithm takes as input the security parameter λ (in unary), an integer bound T (in binary) and k messages $(m_1, \ldots, m_k)$, where each $m_i \in \{0,1\}^{\ell(\lambda)}$ and k is some polynomial in λ. It outputs public parameters $\mathsf{PP}_{\mathsf{ltr}}$ and a state $v_0 \in \mathsf{SplScheme}$.

$\mathsf{Iterate}(\mathsf{PP}_{\mathsf{ltr}}, v_{\mathsf{in}}, m)$ The iterate algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{ltr}}$, a state v_{in}, and a message $m \in \{0,1\}^{\ell(\lambda)}$. It outputs a state $v_{\mathsf{out}} \in \mathsf{SplScheme}_\lambda$.

For simplicity of notation, the dependence of ℓ on λ will not be explicitly mentioned. Also, for any integer $k \leq T$, we will use the notation $\mathsf{Iterate}^k(\mathsf{PP}_{\mathsf{ltr}}, v_0, (m_1, \ldots, m_k))$ to denote $\mathsf{Iterate}(\mathsf{PP}_{\mathsf{ltr}}, v_{k-1}, m_k)$, where $v_j = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{ltr}}, v_{j-1}, m_j)$ for all $1 \leq j \leq k-1$.

A.3 Splittable Signatures

We describe the syntax of the splittable signatures scheme below.

Syntax A splittable signature scheme $\mathsf{SplScheme}$ for message space Msg consists of the following algorithms:

$\mathsf{SetupSpl}(1^\lambda)$ The setup algorithm is a randomized algorithm that takes as input the security parameter λ and outputs a signing key SK, a verification key VK and *reject-verification key* $\mathsf{VK}_{\mathsf{rej}}$.

$\mathsf{SignSpl}(\mathsf{SK}, m)$ The signing algorithm is a deterministic algorithm that takes as input a signing key SK and a message $m \in \mathsf{Msg}$. It outputs a signature σ.

$\mathsf{VerSpl}(\mathsf{VK}, m, \sigma)$ The verification algorithm is a deterministic algorithm that takes as input a verification key VK, signature σ and a message m. It outputs either 0 or 1.

$\mathsf{SplitSpl}(\mathsf{SK}, m^*)$ The splitting algorithm is randomized. It takes as input a secret key SK and a message $m^* \in \mathsf{Msg}$. It outputs a signature $\sigma_{\mathsf{one}} = \mathsf{SignSpl}(\mathsf{SK}, m^*)$, a one-message verification key $\mathsf{VK}_{\mathsf{one}}$, an all-but-one signing key $\mathsf{SK}_{\mathsf{abo}}$ and an all-but-one verification key $\mathsf{VK}_{\mathsf{abo}}$.

$\mathsf{SignSplAbo}(\mathsf{SK}_{\mathsf{abo}}, m)$ The all-but-one signing algorithm is deterministic. It takes as input an all-but-one signing key $\mathsf{SK}_{\mathsf{abo}}$ and a message m, and outputs a signature σ.

KLW described various security notions corresponding to the above splittable signatures scheme. We describe only one of the properties that will be useful for this work. This security notion is termed as $\mathsf{VK}_{\mathsf{one}}$ indistinguishability and states that given a signature on a message m, an adversary should not be able to distinguish the verification key VK from the split verification key $\mathsf{VK}_{\mathsf{one}}$, that is computed as a result of applying $\mathsf{SplitSpl}$ on the signing key and message m.