Functional Encryption for Turing Machines

Prabhanjan Ananth[*]
Department of Computer Science and
Center for Encrypted Functionalities,
UCLA, USA
prabhanjan.va@gmail.com

Amit Sahai[†]
Department of Computer Science and
Center for Encrypted Functionalities,
UCLA, USA
sahai@cs.ucla.edu

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.

[*]This work was partially supported by a grant from the Simons Foundation (#360584 to Prabhanjan Ananth).

[†]Research supported in part from a DARPA/ONR PROCEED award, 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 U.S. Office of Naval Research under Contract N00014-11-1-0389. 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.

Contents

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 [SW05] and formalized later in the works of [BSW11, O'N10], has emerged as a powerful form of encryption well-suited to many contemporary applications (see [BSW11, BSW12] for further discussion of application scenarios for functional encryption). A functional encryption (FE) scheme allows a user possessing a key associated 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 [BSW11, AGVW13]; however, there are sound adaptive indistinguishability-based formulations [BSW11] that also imply simulation-based security in restricted settings [CIJ$^+$13]. Following most recent work on FE [GGH$^+$13, Wat15, GGHZ14, ABSV15], 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 [KSW08]. 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 [SS10, GVW12], but only for functions represented by circuits. A major advance occurred in the work of [GGH$^+$13], 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, [GGH$^+$13] and other FE results deal with functionalities represented by *circuits* – and representing functions as circuits gives rise to two major drawbacks. The first drawback is that a circuit representation takes the worst case running time on every input. Research to deal with this issue was initiated by Goldwasser et al. [GKP$^+$13], and there have been several recent works [BGL$^+$15, CHJV15, KLW15, CCC$^+$15], 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 [BGL$^+$15, CHJV15, KLW15] construct iO for Turing Machines only with bounded input length, where the bound must be specified at the time of obfuscating the Turing Machine. If this iO method is combined, for example, with the FE construction recipe of [GGH$^+$13], then this would only yield FE for functions with a bound on input length specified at the time of setting up the FE scheme.
There have been works [BCP14, IPS15] on overcoming the issue of a priori bounded input lengths but these are based on strong knowledge-type assumptions called differing inputs obfuscation [BGI$^+$12, BCP14, ABG$^+$13] or more recently public-coin differing inputs obfuscation [IPS15]. 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 an adaptively secure FE scheme for Turing machines assuming the existence of indistinguishability obfuscation and one-way functions.*

[1]We stress that despite recent cryptanalytic progress, iO candidates such as [BGK$^+$14] remain beyond the reach of any known cryptanalytic technique.

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 [BHR12, AIKW15] or under restricted settings [BGG+14].

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 [SS10] and Gorbunov et al. [GVW12] 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 [BGL+15, CHJV15, KLW15], based on iO for circuits.

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 [AIK06].
- 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 [Yao86]. 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. [BHR12], 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 [KLW15]. 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 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 $\mathsf{combiner}$ 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 $\mathsf{combiner}$: 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

[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.

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 [GGG+14] 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 [Wat15], later revisited by [ABSV15], in the context of constructing adaptively secure FE for circuits.

Our boosting mechanism, however, diverges in several ways from the previous works of [Wat15, ABSV15]. 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, [ABSV15] 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 [BS14] 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.

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.

The standard cryptographic notions of pseudorandom functions and symmetric encryption schemes are defined in Appendix B.1. We also recall the definition of Turing machines in the same section. 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.

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 is modeled in the form of a game between the challenger and an (efficient) adversary. The adversary is allowed to request for an arbitrary number of functional keys from the challenger. In addition, the adversary can also submit message pairs of the form (m_0, m_1), where $|m_0| = |m_1|$ and in return it receives a ciphertext of m_b, where $b \xleftarrow{\$} \{0,1\}$ is sampled by the challenger. At the end of the game, the adversary outputs a bit b'. The adversary wins the game if $b' = b$ and if $f(m_0) = f(m_1)$, for all Turing machine queries f, message pair queries (m_0, m_1). Recall that, we only consider Turing machines which also output the time taken to execute and hence, $f(m_0) = f(m_1)$ ensures that the running time of f on m_0 and m_1 are the same.

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.*

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

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 Appendix B.3.

2.2 (Compact) FE for circuits

2.2.1 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 [BSW11, O'N10].

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 [AJ15, BV15] 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*

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

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{FE}}(1^\lambda, b)$*, 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.

2.2.2 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 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 ($\mathsf{PrivFE.Setup}$, $\mathsf{PrivFE.KeyGen}$, $\mathsf{PrivFE.Enc}$, $\mathsf{PrivFE.Dec}$). 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 [BS14]. In the security game of function privacy, a function query made by the adversary is a pair of functions and in response it receives a functional key corresponding to either of the two functions. As long as both the functions are such that they do not split the challenge message-pairs, the adversary should not be able to tell which function was used to generate the functional key. That is, the output of the left function on the left message should be the same as the output of the right function on the right message.

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* $\mathsf{PubFE} = (\mathsf{PrivFE.Setup}, \mathsf{PrivFE.KeyGen}, \mathsf{PrivFE.Enc}, \mathsf{PrivFE.Dec})$ *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*

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

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PubFE}}(1^\lambda, b)$*, 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 [SS10, GVW12, GKP+12] 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 below.

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.

Definition 4 (Single-ciphertext Single-key Private-key FE for TMs)**.** *A private-key functional encryption scheme* $\mathsf{OneCTKey} = (\mathsf{OneCTKey.Setup}, \mathsf{OneCTKey.KeyGen}, \mathsf{OneCTKey.Enc}, \mathsf{OneCTKey.Dec})$ *for Turing machines 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 single-ciphertext***

single-key private-key FE scheme for Turing machines *if for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mu(\lambda)$ such that the advantage of $\mathcal{A}$ is defined to be*

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

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

1. *The challenger first executes $\mathsf{OneCTKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneCTKey.MSK}$.*
2. ***Query Phase I**: The adversary can submit a single Turing machine query f to the challenger during this phase. The challenger generates $\mathsf{OneCTKey}.sk_f$, where $\mathsf{OneCTKey}.sk_f$ is the output of $\mathsf{OneCTKey.KeyGen}($ $\mathsf{OneCTKey.MSK}, f)$. It then sends $\mathsf{OneCTKey}.sk_f$ to the adversary. We emphasize that the adversary can only make at most one function query in this phase.*
3. ***Challenge Phase**: The adversary submits a message pair (m_0, m_1) to the challenger. The challenger generates $\mathsf{CT} = \mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m_b)$. If $f(m_0) = f(m_1)$ for a Turing machine query f made in Query Phase I, the challenger sends CT to the adversary. Otherwise, it aborts. We emphasize that the adversary can make at most one message query in this phase.*
4. ***Query Phase II**: If the adversary has already made a Turing machine query during Query Phase I then the adversary can make no more Turing machine queries during this phase. Otherwise, the adversary can submit a Turing machine query f to the challenger during this phase. The challenger generates $\mathsf{OneCTKey}.sk_f$, where $\mathsf{OneCTKey}.sk_f$ is the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f)$. If $f(m_0) = f(m_1)$, for the message query (m_0, m_1), it sends $\mathsf{OneCTKey}.sk_f$ to the adversary and otherwise, it aborts. As in Query Phase I, we emphasize that the adversary can only make at most one Turing machine query in this phase.*
5. *The output of the experiment is b', where b' is the output of $\mathcal{A}$.*

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) [GGG+14] 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.

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

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 $\mathsf{2FE.CT}_y \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, y_0, 0)$ if $b = 0$ else it responds with $\mathsf{2FE.CT}_y \leftarrow \mathsf{2FE.Enc}(\mathsf{2FE.MSK}, y_1, 0)$.
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 5. *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.

$\underline{\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f \in \mathcal{F})}$: 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 $\mathsf{OneCTKey}.sk_M = (\mathsf{2FE.CT}_M, \mathsf{2FE.sk} \oplus R)$.

$\underline{\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}, m)}$: 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.

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

Proof. We prove the correctness, efficiency and adaptive security properties below.

Correctness. Suppose the master secret key $\mathsf{OneCTKey.MSK} = \mathsf{SRE.sk}$ is the output of $\mathsf{OneCTKey.Setup}(1^\lambda)$. And let the ciphertext $\mathsf{OneCTKey.CT}$, parsed as $(\widetilde{m}, R)$, be the output of $\mathsf{OneCTKey.Enc}$ on input $(\mathsf{OneCTKey.MSK}, m)$. Further, let the functional key $\mathsf{OneCTKey}.sk_f$, parsed as $(\widetilde{f}, S)$, be the output of $\mathsf{OneCTKey.KeyGen}$, where $S = \widetilde{\mathsf{UTM}_\lambda} \oplus R$ with $\widetilde{\mathsf{UTM}_\lambda}$ being the output of $\mathsf{SRE.TMEnc}(\mathsf{SRE.sk}, \mathsf{UTM}_\lambda)$. Then, from the correctness property of the 2-stage SRE, we have the output of the decode algorithm $\mathsf{SRE.Decode}$ on input $(\widetilde{\mathsf{UTM}_\lambda}, \widetilde{f}, \widetilde{m})$ to be $f(m)$. Thus, we have the output of the decryption algorithm $\mathsf{OneCTKey.Dec}$ on input $(\mathsf{OneCTKey}.sk_f, \mathsf{OneCTKey.CT})$, which is nothing but the output of the decode procedure $\mathsf{SRE.Decode}(\widetilde{\mathsf{UTM}_\lambda}, \widetilde{f}, \widetilde{m})$, to be $f(m)$.

Efficiency. We need to argue about the efficiency of the key generation, encryption as well as the decryption procedures. We first argue that the running time of the key generation algorithm, $\mathsf{OneCTKey.KeyGen}$, on input master secret key $\mathsf{OneCTKey.MSK} = (\mathsf{SRE.sk}, R)$ and function f, is a polynomial in $(\lambda, |f|)$. We first note that the master secret key $\mathsf{OneCTKey.MSK}$ itself is a fixed polynomial in the security parameter λ. The running time of the input encoding algorithm $\mathsf{SRE.InpEnc}$ on input $(\mathsf{SRE.sk}, f, 1)$ is a polynomial in $(\lambda, |f|)$ and the running time of the TM encode algorithm $\mathsf{SRE.TMEnc}$ on input $(\mathsf{SRE.sk}, \mathsf{UTM})$ is a polynomial in λ – this follows from the efficiency properties of the 2-stage SRE scheme. Thus, the running time of $\mathsf{OneCTKey.KeyGen}$ is a polynomial in $(\lambda, |f|)$.

We now analyze the running time of the encryption algorithm $\mathsf{OneCTKey.Enc}$, on input master secret key $\mathsf{OneCTKey.MSK} = (\mathsf{SRE.sk}, R)$ and message m. The running time of the input encoding algorithm $\mathsf{SRE.InpEnc}$

on input $(\mathsf{SRE.sk}, m, 2)$, is a polynomial in $(\lambda, |m|)$ – this again follows from the efficiency of the input encoding algorithm. Thus, the running time of $\mathsf{OneCTKey.Enc}$ is a polynomial in $(\lambda, |m|)$.

Finally, we analyze the running time of the decryption algorithm $\mathsf{OneCTKey.Dec}$, on input the functional key of f, $\mathsf{OneCTKey}.sk_f = (\widetilde{f}, S)$ and ciphertext of message m, $\mathsf{OneCTKey.CT} = (\widetilde{m}, R)$, where $S = \widetilde{\mathsf{UTM}_\lambda} \oplus R$. To determine this quantity, it suffices to analyze the running time of the decoding procedure $\mathsf{SRE.Decode}$ on input $(\widetilde{\mathsf{UTM}_\lambda}, \widetilde{f}, \widetilde{m})$, which is nothing but a polynomial in $(\lambda, |\mathsf{UTM}_\lambda|, |f|, |x|, t)$, where t is the running time of the TM f on input x, as desired.

Adaptive Security. We need to argue that the FE scheme $\mathsf{OneCTKey}$ that we constructed above is indeed an adaptively secure single-ciphertext single-key FE. More formally, we prove the following. The proof of the theorem essentially follows from the security of the 2-stage SRE scheme.

Lemma 1. *Assuming the security of 2-stage SRE* SRE*, the scheme* $\mathsf{OneCTKey}$ *is adaptively secure.*

Proof. Suppose $\mathsf{OneCTKey}$ is not adaptively secure, that is, there exists a PPT adversary $\mathcal{A}$ that breaks the security of $\mathsf{OneCTKey}$. We then design a reduction $\mathcal{B}$ that internally runs the adversary $\mathcal{A}$ and breaks the security of SRE.

The reduction $\mathcal{B}$ essentially simulates the role of the challenger in the security game of $\mathsf{OneCTKey}$ and at the same time takes the role of the adversary in the game of SRE. Depending on the order in which the adversary makes the function and the message query, there are two cases.

1. **Adversary first submits a TM query:** Suppose the adversary submits the TM query f to the $\mathcal{B}$. The reduction then represents f as a string and then submits this as an input query to the challenger of SRE. In return it receives $\widetilde{f}$. It then sends the functional key $(\widetilde{f}, S')$ to $\mathcal{A}$, where S' is a random string in $\{0,1\}^{\ell_{\mathsf{UTM}}}$. The reduction then receives the input pair (m_0, m_1) from the adversary $\mathcal{A}$. The reduction then submits the input query (m_0, m_1) to the challenger of SRE. In response to this query, it receives the encoding $\widetilde{m}^*$. Then, $\mathcal{B}$ sends the universal Turing machine UTM_λ as a TM query to the challenger and it receives the TM encoding $\widetilde{\mathsf{UTM}_\lambda}$. The reduction then sends the challenge ciphertext $(\widetilde{m}^*, R')$ to $\mathcal{A}$, where R' is set to be $S \oplus \widetilde{\mathsf{UTM}_\lambda}$.
2. **Adversary first submits a message query:** Suppose the adversary submits the message pair query (m_0, m_1) to $\mathcal{B}$. The reduction $\mathcal{B}$ then forwards this message query (m_0, m_1) to the challenger of SRE. In return it receives $\widetilde{m}^*$. The reduction then sends the challenge ciphertext $(\widetilde{m}^*, R)$ to the adversary $\mathcal{A}$, where R is a random string of length $\{0,1\}^{\ell_{\mathsf{UTM}}}$. The adversary then sends the Turing machine f to $\mathcal{B}$, who then forwards it as an input query to the challenger of SRE. The reduction $\mathcal{B}$ receives back the input encoding $\widetilde{f}$ – this won't be forwarded to $\mathcal{A}$ at this point. The reduction $\mathcal{B}$ then submits, as TM query, the universal Turing machine UTM_λ, to the challenger of SRE. In response, it receives $\widetilde{\mathsf{UTM}_\lambda}$. The reduction then composes the response to the TM query of $\mathcal{A}$. It sends the functional key $(\widetilde{f}, \widetilde{\mathsf{UTM}_\lambda})$ to $\mathcal{A}$.

In both the cases, we note that the distribution of the functional key and the challenge ciphertext to be identical as produced by $\mathcal{B}$ is identical to the distribution as produced according to the scheme. Thus, if $\mathcal{A}$ breaks the security of $\mathsf{OneCTKey}$ with non-negligible probability then $\mathcal{B}$ breaks the security of SRE with non-negligible probability. □

□

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 [KLW15]. 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 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.

[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.

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 Section 3.4 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.

3.4 Constructing Semi-Adaptive 2-ary FE for TMs: Formal Description

The randomized encodings construction of [KLW15] is the starting point of our construction. However, to achieve our goal, we need to make several modifications to their construction. To present things in a more modular way, we identify a primitive that is "closest" to the construction of KLW. We call this primitive, *3-stage KLW*. This notion is similar to the notion of 2-ary FE for TMs (in the single-key single-ciphertext) setting except that there is an algorithm called "Combine" that gives a way to combine two encryptions (we call them encodings in the following description) into one ciphertext.

We provide a construction of 3-stage KLW which builds on the RE construction of [KLW15]. Using this primitive, we show how to obtain 2-ary FE for TMs.

3.4.1 3-stage KLW

We first describe the syntax of 3-stage KLW scheme. The scheme we construct is denoted by $\mathsf{3StgKLW}$. It consists of 4 PPT algorithms, namely ($\mathsf{3StgKLW.Setup}$, $\mathsf{3StgKLW.Encode}$, $\mathsf{3StgKLW.Combine}$, $\mathsf{3StgKLW.TMEncode}$, $\mathsf{3StgKLW.Decode}$).

- **Setup algorithm,** $\mathsf{3StgKLW.Setup}(1^\lambda)$: It takes as input security parameter λ and it outputs the secret parameters $\mathsf{3StgKLW.sk}$.
- **Input encoding algorithm,** $\mathsf{3StgKLW.Encode}(\mathsf{3StgKLW.sk}, x, b)$: It takes as input secret parameters $\mathsf{3StgKLW.sk}$, input x, bit b and outputs an encoding $\widetilde{x} = (\widehat{x}, \mathsf{acc}_x)$.
- **Combiner algorithm,** $\mathsf{3StgKLW.Combine}(\mathsf{3StgKLW.sk}, \mathsf{acc}_x, \mathsf{acc}_y)$: It takes as input secret parameters $\mathsf{3StgKLW.sk}$, accumulator values acc_x, acc_y, and outputs joint accumulator value $\mathsf{acc}_{x||y}$.
- **TM encoding algorithm**, $\mathsf{3StgKLW.TMEncode}(\mathsf{3StgKLW.sk}, M)$: It takes as input secret parameters $\mathsf{3StgKLW.sk}$, Turing machine M and outputs an encoding $\widetilde{M}$.
- **Decode algorithm**, $\mathsf{3StgKLW.Decode}(\widehat{x}, \widehat{y}, \mathsf{acc}_{x||y}, \widetilde{M})$: It takes as input, encodings $\widehat{x}$, $\widehat{y}$, joint accumulator values $\mathsf{acc}_{x||y}$, TM encoding $\widetilde{M}$, and outputs z.

The above scheme is required to satisfy the correctness, efficiency and security properties.

Correctness. Suppose the output of $\mathsf{3StgKLW.Encode}(\mathsf{3StgKLW.sk}, x, 0)$ is $\widetilde{x} = (\widehat{x}, \mathsf{acc}_x)$ and the output of $\mathsf{3StgKLW.Encode}(\mathsf{3StgKLW.sk}, y, 1)$ is $\widetilde{y} = (\widehat{y}, \mathsf{acc}_y)$. And let the output of the combine algorithm $\mathsf{3StgKLW.Combine}$ on input $(\mathsf{3StgKLW.sk}, \mathsf{acc}_x, \mathsf{acc}_y)$ be $\mathsf{acc}_{x||y}$. Further let the output of $\mathsf{3StgKLW.TMEncode}(\mathsf{3StgKLW.sk}, M)$ be the TM encoding $\widetilde{M}$. Then the output of the decode algorithm $\mathsf{3StgKLW.Decode}$ on input $(\widehat{x}, \widehat{y}_1, \mathsf{acc}_{x||y}, \widetilde{M})$ is always $M(x, y)$.

Efficiency. A 3-stage KLW scheme is required to satisfy the following efficiency properties:

- The running time of $\mathsf{3StgKLW.Encode}$ on input $(\mathsf{3StgKLW.sk}, x, b)$ is a polynomial in the security parameter λ and the length of the instance, $|x|$.
- The running time of $\mathsf{3StgKLW.Combine}$ on input $(\mathsf{3StgKLW.sk}, \mathsf{acc}_x, \mathsf{acc}_y)$ is a polynomial in the security parameter λ. In particular, the run time does not depend on the length of x or y that were used to generate the parameters acc_x and acc_y.
- The running time of $\mathsf{3StgKLW.TMEncode}$ on input $(\mathsf{3StgKLW.sk}, M)$ is a polynomial in the security parameter λ and size of the Turing machine, $|M|$.
- The running time of $\mathsf{3StgKLW.Decode}$ on input $(\widetilde{x}, \widetilde{y}, \mathsf{acc}_{x||y}, \widetilde{M})$ is a polynomial in the security parameter $\lambda, |M|, |x|$ and t, where t is the running time of M on input (x, y).

Semi-Adaptive Security. The above 3-stage KLW scheme satisfies the following semi-adaptive security property. This notion is identical to the semi-adaptive security notion defined in the context of 2-ary FE for TMs. In the game, adversary can request for input encodings, denoted by (x_0, y_0) and (x_1, y_1), in an adaptive manner. The response to these queries are made by the challenger by encoding x_b and y_b, where the bit b is picked at random. After this, the challenger sends the joint accumulator value (of encodings of x and y) to the adversary. The adversary then queries for a TM M to the challenger who then responds with an encoding of M. The game ends with the adversary guessing the bit b.

The game is formally described below. We denote by $\mathcal{A}$, the adversary in the experiment below.

$\underline{\mathsf{Expt}_{\mathsf{3StgKLW},\mathcal{A}}(1^\lambda, b)}$:

1. The challenger runs $\mathsf{3StgKLW.Setup}(1^\lambda)$ to obtain $\mathsf{3StgKLW.sk}$. It then picks a challenge 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 $(\widetilde{x}_0, \mathsf{acc}_{x_0})$ if $b = 0$ else it responds with $(\widetilde{x}_1, \mathsf{acc}_{x_1})$.
   - The adversary submits the message query (y_0, y_1), corresponding to 1^{st} position to the challenger. The challenger responds with $(\widetilde{y}_0, \mathsf{acc}_{y_0})$ if $b = 0$ else it responds with $(\widetilde{y}_1, \mathsf{acc}_{y_1})$.
3. The challenger then sends the combined encodings, $\mathsf{acc}_{x_b||y_b}$ to the adversary.
4. The adversary then submits the Turing machine query M to the challenger. The challenger first checks if $M(x_0, y_0) = M(x_1, y_1)$. If the check does not go through, it aborts. Otherwise, the challenger sends the TM encoding $\widetilde{M}$ to the adversary.
5. The adversary outputs bit b'.

We say that the adversary wins the game $\mathsf{Expt}_{\mathsf{3StgKLW}}(1^\lambda)$ if $b' = b$.

Definition 6. *The 3-stage KLW scheme is secure if the adversary wins in* $\mathsf{Expt}_{\mathsf{3StgKLW},\mathcal{A}}(1^\lambda, b)$ *with probability at most* $1/2 + \mathsf{negl}(\lambda)$.

3.4.2 Construction of 3-stage KLW

The starting point of our construction is the succinct randomized encodings[5] construction of [KLW15]. In general, it is not clear how to generically transform a succinct randomized encodings scheme into a 2-stage scheme but we will make use of the special structure satisfied by the construction of [KLW15] to achieve our goal. We describe the encoding scheme of Koppula et al. at a high level below.

Structure of Koppula et al. [KLW15]. Let the input to the encoding procedure be the Turing machine M and input x.

1. The encoding scheme begins by generating the encryption and the decryption keys of a public-key encryption scheme. It then encrypts the input x, bit by bit, using the public key encryption scheme, to obtain a tuple of ciphertexts.
2. The encoding scheme then initializes the parameters of the accumulator and the iterator scheme. This is done using the accumulator setup SetupAcc (Refer to 2) and iterator setup SetupItr (Refer to 2). Positional accumulators allow for compressing work tape into a small accumulator value just like a collision-resilient hash function. However, unlike a traditional collision-resilient hash function, it is possible to program the accumulator such that a hash value information theoretically binds a tape symbol.
3. The ciphertexts computed in Bullet 1 are then accumulated using the storage of the accumulator. This process is carried out using the helper algorithms of the accumulator scheme. The resulting storage that is in the form of a tree has an associated accumulator value, which is nothing but the root of the tree. This storage will be part of the encoding.
4. The accumulator value along with the iterator value and the initial state is then signed using a splittable signature scheme.
5. Finally, a program is designed that computes the next step function of the Turing machine M, that is to be encoded. This program also performs additional checks to ensure that a malicious adversary does not input incorrect values. These checks are carried out using accumulator, iterator and splittable signatures. In order to hide these checks, the program is then obfuscated. This completes the encoding process.

We first make some observations about the structure of the KLW scheme as stated above. The first observation is that the storage computed in Bullet 3 need not be part of the encoding. This is because this storage can be recomputed by the decoding algorithm using the ciphertexts and the public parameters of the accumulator scheme. The second observation is that the Bullets 1-4 do not depend on the Turing machine M to be encoded and at the same time, Bullet 5 does not depend on the input x. Using this observation, we can split the encoding procedure into two sub procedures, one for input encoding of x and the other for TM encoding of M.

In addition to the above simplifications, we make one modification to the structure of the scheme. Recall that in a 2-ary FE scheme, we should be able to split the input into two parts with the ability to encode each part separately and then there should be a mechanism to combine both these encodings. To do this, we should be able to split the storage, handled by the accumulator scheme, and later be able to join the storage together. Fortunately, the accumulator scheme constructed by KLW already has this property. The storage in the scheme of KLW is in the form of a tree. Joining two storage components paramounts to joining two trees into one tree with a new root node.

We now formally describe the scheme that incorporates the simplifications and the modification described above. We denote the 3-stage KLW scheme we construct to be 3StgKLW. The tools used in the below construction are as follows: an indistinguishability obfuscator iO for polynomial sized circuits, accumulator scheme (SetupAcc, EnforceRead, EnforceWrite, PrepRead, PrepWrite, VerifyRead, WriteStore, Update), iterator scheme (SetupItr,ItrEnforce, Iterate) and a splittable signature scheme represented by (SetupSpl, SignSpl, VerSpl, SplitSpl, SignSplAbo). In addition we use a puncturable pseudorandom function family and a public key encryption scheme, (PKE.Setup, PKE.Enc, PKE.Dec).

[5]We note that the construction of Koppula et al. was proven secure with respect to a notion that is termed as machine hiding encodings. However, as observed in Section 7.3 (of their ePrint version), the same construction can be shown to be a secure succinct randomized encoding.

$\underline{\mathsf{3StgKLW.Setup}(1^\lambda)}$: It first samples puncturable PRF keys K_E and K_A. K_E will be used for computing an encryption of the symbol and state, and K_A to compute the secret key/verification key for signature scheme. Let $(r_{0,1}, r_{0,2}, r_{0,3}) = \mathsf{PRF}(K_E, 0)$. It then executes the following:

- *PKE setup.* Generate $(\mathsf{pk}, \mathsf{sk}) = \mathsf{PKE.Setup}(1^\lambda; r_{0,1})$. It computes $\mathsf{CT_{st}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, q_0)$.
- *Accumulator setup.* Compute $(\mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0) \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$.
- *Iterator setup.* Compute $(\mathsf{PP_{Itr}}, v_0) \leftarrow \mathsf{SetupItr}(1^\lambda, T)$.

Finally, output the secret key $\mathsf{3StgKLW.sk} = (K_E, K_A, \mathsf{PP_{Acc}}, \mathsf{PP_{Itr}}, v_0, \mathsf{CT_{st}})$.

$\underline{\mathsf{3StgKLW.Encode}(\mathsf{3StgKLW.sk}, x, b)}$: Parse the input secret key $\mathsf{3StgKLW.sk}$ as $(K_E, K_A, \mathsf{PP_{Acc}}, \mathsf{PP_{Itr}}, v_0, \mathsf{CT_{st}})$. It then computes the following. Let $\ell_{\mathsf{Inp}} = |x|$.

- *Encrypt the input tape.* It encrypts each bit of x separately; that is, it computes $\mathsf{CT}_i = \mathsf{PKE.Enc}(\mathsf{pk}, x_i)$ for $1 \leq i \leq \ell_{\mathsf{Inp}}$. Denote by CT_x to be the tuple $(\mathsf{CT}_1, \ldots, \mathsf{CT}_{\ell_{\mathsf{Inp}}})$.
- *Compute the storage tree on the input tape.* These ciphertexts are 'accumulated' using the accumulator. It computes $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP_{Acc}}, \widetilde{store}_{j-1}, j-1, (\mathsf{CT}_j, 0))$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP_{Acc}}, \widetilde{store}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP_{Acc}}, \widetilde{w}_{j-1}, \mathsf{Inp}_j, j-1, aux_j)$ for $1 \leq j \leq \ell_{\mathsf{Inp}}$. It sets $w_x = \widetilde{w}_{\ell_{\mathsf{Inp}}}$ and $s_x = \widetilde{store}_{\ell_{\mathsf{Inp}}}$.

Output $\widetilde{x} = (\widehat{x} = \mathsf{CT}_x, \mathsf{acc}_x = w_x)$.

$\underline{\mathsf{3StgKLW.Combine}(\mathsf{3StgKLW.sk}, \mathsf{acc}_x = w_x, \mathsf{acc}_y = w_y)}$: Parse the input secret key $\mathsf{3StgKLW.sk}$ as $(K_E, K_A, \mathsf{PP_{Acc}}, \mathsf{PP_{Itr}}, v_0, \mathsf{CT_{st}})$. Then it does the following operation:

- *Combine both the accumulator values w_x and w_y to obtain w_0*: the public parameter $\mathsf{PP_{Acc}}$ is first parsed as a program P. Then execute the program $P(w_x, w_y)$ to obtain w_0. This will be the joint accumulator of x_0 and x_1. Let $r_A = F(K_A, 0)$, $(\mathsf{SK}_A, \mathsf{VK}_A) = \mathsf{SetupSpl}(1^\lambda; r_A)$ and $\sigma_0 = \mathsf{SignSpl}(\mathsf{SK}_A, (v_0, \mathsf{CT_{st}}, w_0, 0))$.

Output $\mathsf{acc}_{x||y} = (w_0, \sigma_0)$.

$\underline{\mathsf{3StgKLW.TMEncode}(\mathsf{3StgKLW.sk}, M')}$: On input $(\mathsf{3StgKLW.sk}, M')$, first parse $\mathsf{3StgKLW.sk}$ as $(K_E, K_A, \mathsf{PP_{Acc}}, \mathsf{PP_{Itr}}, v_0, \mathsf{CT_{st}})$. It first transforms M' into an oblivious TM M. It computes an obfuscation $P \leftarrow \mathsf{iO}(\mathsf{Prog}\{M, T, \mathsf{PP_{Acc}}, \mathsf{PP_{Itr}}, K_E, K_A\})$ where Prog is defined in Figure 1. It outputs the machine encoding $\widetilde{M} = P$.

$\underline{\mathsf{3StgKLW.Decode}(\widehat{x}, \widehat{y}, \mathsf{acc}_{x||y}, \widetilde{M})}$: The input to $\mathsf{SRE.Decode}$ are input encodings $\widehat{x} = \mathsf{CT}_x$ and $\widehat{y} = \mathsf{CT}_y$, along with their combined encoding $\mathsf{acc}_{x||y} = (w_0, \sigma_0)$.

- *Initialize the storage on CT_x and CT_y.* Compute $\widetilde{store}_{x,j} = \mathsf{WriteStore}(\mathsf{PP_{Acc}}, \widetilde{store}_{x,j-1}, j-1, (\mathsf{CT}_j x, j, 0))$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP_{Acc}}, \widetilde{store}_{x,j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP_{Acc}}, \widetilde{w}_{x,j-1}, \mathsf{Inp}_j, j-1, aux_j)$ for $1 \leq j \leq \ell_{\mathsf{Inp}}$. Finally, it sets $w_x^* = \widetilde{w}_{x,\ell_{\mathsf{Inp}}}$ and $s_x = \widetilde{store}_{\ell_{\mathsf{Inp}}}$. Correspondingly, initialize storage on the ciphertext CT_y as well. We denote the resulting accumulator value to be w_y^* and denote the storage to be s_y.
- *Merging the storage tree.* Assign the final storage, corresponding to the joint value $(x||y)$ to be s_0, which is obtained by merging the trees s_x and s_y with the root of the final storage s_0 being w_0.

Once this is computed, run the decode algorithm $\mathsf{Mc.Dec}$ of [KLW15] to recover the output. That is, execute $\mathsf{Mc.Dec}(P, w_0, v_0, \sigma_0, s_0)$ to obtain the value out. Output the value out.

The correctness, efficiency, and the security properties of the above scheme $\mathsf{3StgKLW}$ essentially follows along the same lines as the security proof of machine hiding encodings scheme of [KLW15]. The minor modifications is just incorporating the following two observations (i) that their proof also works when the adversary can query the input to obtain the storage tree and only after that he can query for the TM and, (ii) the storage tree need not necessarily be balanced as in the work of KLW. Thus we have,

Theorem 3. *Assuming the existence of indistinguishability obfuscation for circuits and public key encryption schemes,* $\mathsf{3StgKLW}$ *is a secure 3-stage KLW scheme.*

```
Program Prog

Constants: Turing machine $M = \langle Q, \Sigma_{\mathrm{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}} \rangle$, Public parameters for accumulator $\mathsf{PP}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{PP}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{CT}_{\mathsf{sym,in}}, \mathsf{lw})$, encrypted state $\mathsf{CT}_{\mathsf{st,in}}$, accumulator value $w_{\mathsf{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathsf{in}}$, accumulator proof $\pi$, auxiliary value $aux$.

1. Let $\mathsf{pos}_{\mathsf{in}} = \mathsf{tmf}(t-1)$ and $\mathsf{pos}_{\mathsf{out}} = \mathsf{tmf}(t)$, where $\mathsf{tmf}$ is the time movement function that determines the current position of the tape head given the current time step.
2. Verification step.
    • If $\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_{\mathsf{in}}, (\mathsf{CT}_{\mathsf{sym,in}}, \mathsf{lw}), \mathsf{pos}_{\mathsf{in}}, \pi) = 0$ or $\mathsf{lw} \geq t$, output $\perp$.
    • Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathsf{SK}_A, \mathsf{VK}_A, \mathsf{VK}_{A,\mathsf{rej}}) = \mathsf{SetupSpl}(1^\lambda; r_{S,A})$.
    • Let $m_{\mathsf{in}} = (v_{\mathsf{in}}, \mathsf{CT}_{\mathsf{st,in}}, w_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}})$. If $\mathsf{VerSpl}(\mathsf{VK}_A, m_{\mathsf{in}}, \sigma_{\mathsf{in}}) = 0$ output $\perp$.
3. Decrypting the encrypted tape symbols.
    • Let $(r_{\mathsf{lw},1}, r_{\mathsf{lw},2}, r_{\mathsf{lw},3}) = F(K_E, \mathsf{lw})$, $(\mathsf{pk}_{\mathsf{lw}}, \mathsf{sk}_{\mathsf{lw}}) = \mathsf{PKE.Setup}(1^\lambda; r_{\mathsf{lw},1})$, $\mathsf{sym} = \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{lw}}, \mathsf{CT}_{\mathsf{sym,in}})$.
    • Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $(\mathsf{pk}_{\mathsf{st}}, \mathsf{sk}_{\mathsf{st}}) = \mathsf{PKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{CT}_{\mathsf{st,in}})$.
4. Computing the next message function.
    • Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym})$.
    • If $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{rej}}$ output 0.
    • If $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{acc}}$ output 1.
5. Encrypting the new tape symbol.
    • Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $(\mathsf{pk}', \mathsf{sk}') = \mathsf{PKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{CT}_{\mathsf{sym,out}} = \mathsf{PKE.Enc}(\mathsf{pk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{CT}_{\mathsf{st,out}} = \mathsf{PKE.Enc}(\mathsf{pk}', \mathsf{st}'; r_{t,3})$.
6. Authenticate the computed encrypted tape symbol.
    • Compute $w_{\mathsf{out}} = \mathsf{Accumulate}(\mathsf{PP}_{\mathsf{Acc}}, w_{\mathsf{in}}, (\mathsf{CT}_{\mathsf{sym,out}}, t), \mathsf{pos}_{\mathsf{in}}, aux)$. If $w_{\mathsf{out}} = Reject$, output $\perp$.
    • Compute $v_{\mathsf{out}} = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{CT}_{\mathsf{st,in}}, w_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}}))$.
    • Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathsf{SK}'_A, \mathsf{VK}'_A, \mathsf{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda; r'_{S,A})$.
7. Let $m_{\mathsf{out}} = (v_{\mathsf{out}}, \mathsf{CT}_{\mathsf{st,out}}, w_{\mathsf{out}}, \mathsf{pos}_{\mathsf{out}})$ and $\sigma_{\mathsf{out}} = \mathsf{SignSpl}(\mathsf{SK}'_A, m_{\mathsf{out}})$.
8. Output $\mathsf{pos}_{\mathsf{in}}, \mathsf{CT}_{\mathsf{sym,out}}, \mathsf{CT}_{\mathsf{CT,out}}, w_{\mathsf{out}}, v_{\mathsf{out}}, \sigma_{\mathsf{out}}$.
```

Figure 1 Program Prog

3.4.3 2-ary FE for TMs from 3-stage KLW

We now show how to obtain a 2-ary 1-CT 1-Key FE for TMs scheme in the private key setting from a 3-stage KLW scheme. The main goal is to implement the combiner function as part of the encryption itself. But since the combiner function has information about the secret parameters, the description of the program cannot be in the clear. So, we obfuscate the combiner program and include the obfuscated program to be part of one of the two encryptions. However, we still have the adaptivity issue to deal with – the adversary can query for the encryption having the combiner program at the beginning of the game and only after that he can query for the other input encoding. And, it is not clear how to reduce this to the security game of 3-stage KLW. To overcome this scenario, we use a standard trick which has appeared in many works before and more prominently in the work of Bellare et al. [BHR12]: the input encoding which was queried first will have a random value in the place of combiner program and the other encoding will now contain the XOR of the random value with the combiner program. This way the challenger can adjust the combiner program after seeing both the input queries.

The tools used in our construction are a 3-stage KLW scheme, denoted by 3StgKLW = (3StgKLW.Setup, 3StgKLW.Encode, 3StgKLW.Combine, 3StgKLW.TMEncode, 3StgKLW.Decode), an indistinguishability obfuscation scheme iO for polynomial sized circuits and a splittable signatures scheme (SetupSpl, SignSpl, VerSpl, SplitSpl, SignSplAbo). We denote by 2FE, the 2-ary FE scheme.

$\underline{\mathsf{2FE.Setup}(1^\lambda)}$: On input security parameter λ, execute $\mathsf{3StgKLW.Setup}$ on input (1^λ) to obtain $\mathsf{3StgKLW.sk}$. It then executes $\mathsf{SetupSpl}(1^\lambda)$ twice to obtain the signing key-verification key pairs $(\mathsf{SK}_x, \mathsf{VK}_x)$ and $(\mathsf{SK}_y, \mathsf{VK}_y)$. Pick a random string $\mathsf{R} \in \{0,1\}^{\ell_{\mathsf{comb}}}$, where the value of ℓ_{comb} will be clear later. Output the secret parameters $\mathsf{SRE.sk} = (\mathsf{3StgKLW.sk}, \mathsf{R}, \mathsf{SK}_x, \mathsf{VK}_x, \mathsf{SK}_y, \mathsf{VK}_y)$.

$\underline{\mathsf{2FE.Enc}(\mathsf{SRE.sk}, x, b)}$: On input secret parameters $\mathsf{SRE.sk} = (\mathsf{3StgKLW.sk}, \mathsf{R}, \mathsf{SK}_x, \mathsf{VK}_x, \mathsf{SK}_y, \mathsf{VK}_y)$, input x, position bit b, first execute $\mathsf{3StgKLW.Encode}$ on input $(\mathsf{3StgKLW.sk}, x, b)$ to obtain $(\widehat{x}, \mathsf{acc}_x)$. If $b = 0$, sign the message acc_x using the signing key SK_x, by running $\mathsf{SignSpl}(\mathsf{SK}_x, (\mathsf{acc}_x, 0))$ to obtain σ_x. Else if $b = 1$, compute the signature $\sigma_x \leftarrow \mathsf{SignSpl}(\mathsf{SK}_y, (\mathsf{acc}_x, 0))$. Compute an obfuscation $\mathsf{Combiner}$ as an output of $\mathsf{iO}(\mathsf{Prog2}\{\mathsf{VK}_x, \mathsf{VK}_y, \mathsf{3StgKLW.sk}, \perp\})$, where $\mathsf{Prog2}$ is defined in Figure 2. If $b = 0$, set $\mathcal{S}_b$ to be R otherwise, set $\mathcal{S}_b$ to be $\mathsf{R} \oplus \mathsf{Combiner}$. Output $\widetilde{x} = (\widehat{x}, \mathsf{acc}_x, \sigma_x, \mathcal{S}_b)$.

Program $\mathsf{Prog2}$

Constants: Verification keys VK_x, VK_y, secret parameters $\mathsf{SRE.sk}$, value v.

Input: Accumulator-signature pairs $(\mathsf{acc}_x, \sigma_x)$ and $(\mathsf{acc}_y, \sigma_y)$.

1. Check if both $\mathsf{VerSpl}(\mathsf{VK}_x, (\mathsf{acc}_x, 0), \sigma_0)$ and $\mathsf{VerSpl}(\mathsf{VK}_y, (\mathsf{acc}_y, 1), \sigma_1)$ outpute 1. If the check fails, i.e., either one of the verification algorithms output 0, output $\perp$.
2. Execute $\mathsf{3StgKLW.Combine}(\mathsf{SRE.sk}, \mathsf{acc}_x, \mathsf{acc}_y)$ to obtain $\mathsf{acc}_{x||y}$.
3. Output $\mathsf{acc}_{x||y}$.

Figure 2 Program $\mathsf{Prog2}$

$\underline{\mathsf{2FE.KeyGen}(\mathsf{SRE.sk}, M)}$: On input $(\mathsf{SRE.sk} = (\mathsf{3StgKLW.sk}, \mathsf{R}, \mathsf{SK}_x, \mathsf{VK}_x, \mathsf{SK}_y, \mathsf{VK}_y), M)$, it executes $\mathsf{3StgKLW.TMEncode}(\mathsf{3StgKLW.sk}, M)$ to obtain $\widetilde{M}$. Output $\widetilde{M}$.

$\underline{\mathsf{2FE.Dec}(\widetilde{x}, \widetilde{y}, \widetilde{M})}$: On input encodings $\widetilde{x} = (\widehat{x}, \mathsf{acc}_x, \sigma_x, \mathcal{S}_0)$ and $\widetilde{y} = (\widehat{y}, \mathsf{acc}_y, \sigma_y, \mathcal{S}_1)$ and TM encoding $\widetilde{M}$, it does the following. It computes $S_0 \oplus S_1$ to obtain the obfuscated program $\mathsf{Combiner}$. Then, execute $\mathsf{Combiner}$ on input $(\mathsf{acc}_x, \sigma_x, \mathsf{acc}_y, \sigma_y)$ to obtain $\mathsf{acc}_{x||y}$. It then executes $\mathsf{3StgKLW.Decode}$ on input $(\widehat{x}, \widehat{y}, \mathsf{acc}_{x||y}, \widetilde{M})$ to obtain z. Output z.

We prove the following theorem.

Theorem 4. *Assuming the security of* $\mathsf{3StgKLW}$*, the security of splittable signature scheme* $\mathsf{SplScheme}$*, and the scheme* SRE *satisfies the semi-adaptive security stated in Definition 3.1.*

Proof. We prove the theorem by using the standard hybrid argument technique. The first hybrid $\mathsf{Hybrid}_{1,b}$ corresponds to the real experiment, where the challenger uses the bit b to choose which input to encode. We then, by introducing intermediate hybrids and arguing indistinguishability of every consecutive hybrids using the primitives stated in the theorem, show that $\mathsf{Hybrid}_{1,b}$ is indistinguishable to $\mathsf{Hybrid}_{4,b}$. Finally, we argue that $\mathsf{Hybrid}_{4,0}$ is computationally indistinguishable from $\mathsf{Hybrid}_{4,1}$ which proves that $\mathsf{Hybrid}_{1,0}$ is computationally indistinguishable from $\mathsf{Hybrid}_{1,1}$, as desired.

The hybrids are described below. For convenience sake, we denote the query (x_b, y_b) made by the adversary as $(x_{0,b}, x_{1,b})$.

$\underline{\mathsf{Hybrid}_{1,b} \text{ for } b \in \{0,1\}}$: This corresponds to the real experiment, where the challenger uses the challenge bit b in choosing which input to encode. That is, on receiving an input queries of the form $(x_{0,0}, x_{1,0})$ and $(x_{0,1}, x_{1,1}, 1)$, the challenger responds with input encodings $\widetilde{x}_{0,b}$ and $\widetilde{x}_{1,b}$. The output of the hybrid is the output of the adversary.

$\underline{\mathsf{Hybrid}_{2,b} \text{ for } b \in \{0,1\}}$: In this hybrid, the challenger sets the value of the mask $\mathcal{S}$ in the input encodings depending on the order of the queries. This is unlike the previous hybrid, where the value of mask $\mathcal{S}$ depends on the bit b that is part of the input to the algorithm $\mathsf{3StgKLW.Encode}$.

The challenger produces the secret parameters $(\mathsf{3StgKLW.sk}, \mathsf{R}, \{\mathsf{SK}_c, \mathsf{VK}_c\}_{c \in \{0,1\}})$ by executing $\mathsf{SRE.Gen}(1^\lambda, T)$. If the adversary submits query for the input $((x_{0,0}, x_{0,1}), 0)$ of the form first, the challenger responds by computing an input encoding of $x_{0,b}$, namely $(\widehat{x_{0,b}}, \mathsf{acc}_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}_0)$, computed as in the scheme, except that $\mathcal{S}_0$ is just set to be R, which is part of the secret parameters output by the SRE setup algorithm. Next, when the adversary submits the query $((x_{1,0}, x_{1,1}), 1)$ to the challenger. The challenger then responds with an input encoding of $x_{1,b}$, namely $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_1)$, which are computed as in the scheme, except that $\mathcal{S}_1$ in the encoding is set to be $\mathsf{R} \oplus \mathsf{Combiner}$, where $\mathsf{Combiner}$ is obtained by obfuscating the program $\mathsf{Prog2}\{\mathsf{VK}_x, \mathsf{VK}_y, \mathsf{3StgKLW.sk}, \perp\}$, as described in Figure 2.

Similarly, if the adversary queries for $((x_{1,0}, x_{1,1}), 1)$ then the challenger responds with $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_0)$, where it sets $\mathcal{S}_0$ to be R and when the adversary queries for $((x_{0,0}, x_{0,1}), 0)$, the challenger responds with $(\mathsf{CT}_{x_{0,b}}, w_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}, \mathsf{PP})$, where the string $\mathcal{S}$ is set to be $\mathsf{R} \oplus \mathsf{Combiner}$. The output of the hybrid is the output of the adversary.

The hybrids $\mathsf{Hybrid}_{1,b}$ and $\mathsf{Hybrid}_{2,b}$ are identically distributed.

$\underline{\mathsf{Hybrid}_{3,b} \text{ for } b \in \{0,1\}}$: In this hybrid, the challenger changes the hardwired verification keys in the combiner program such that these verification keys can be used to verify signatures [6] of only one message.

The challenger produces the secret parameters $(\mathsf{3StgKLW.sk}, \mathsf{R}, \{\mathsf{SK}_c, \mathsf{VK}_c\}_{c \in \{0,1\}})$ by executing $\mathsf{SRE.Gen}(1^\lambda, T)$. If the adversary submits query for the input $((x_{0,0}, x_{0,1}), 0)$ of the form first, the challenger responds by computing an input encoding of $x_{0,b}$, namely $(\widehat{x_{0,b}}, \mathsf{acc}_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}_0)$, computed as in the scheme, except that $\mathcal{S}_0$ is just set to be R, which is part of the secret parameters output by the SRE setup algorithm. Next, when the adversary submits the query $((x_{1,0}, x_{1,1}), 1)$ to the challenger. The challenger then responds with an input encoding of $x_{1,b}$, namely $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_1)$, which are computed as in the scheme, except that $\mathcal{S}_1$ in the encoding is set to be $\mathsf{R} \oplus \mathsf{Combiner}$, where $\mathsf{Combiner}$ is obtained by obfuscating the program $\mathsf{Prog2}\{\mathsf{VK}_{\mathsf{one},0}, \mathsf{VK}_{\mathsf{one},1}, \mathsf{3StgKLW.sk}, \perp\}$, as described in Figure 2. Here, $\mathsf{VK}_{\mathsf{one},0}$ (resp., $\mathsf{VK}_{\mathsf{one},1}$), is obtained as part of the output of the execution of $\mathsf{SplitSpl}(\mathsf{SK}_0, (w_x, 0))$(resp., $\mathsf{SplitSpl}(\mathsf{SK}_1, (w_y, 0))$), where $\mathsf{SK}_0, \mathsf{SK}_1$ denote the signing keys that is part of the secret parameters generated by the setup algorithm.

Similarly, if the adversary queries for $((x_{1,0}, x_{1,1}), 1)$ then the challenger responds with $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_0)$, where it sets $\mathcal{S}_0$ to be R and when the adversary queries for $((x_{0,0}, x_{0,1}), 0)$, the challenger responds with $(\mathsf{CT}_{x_{0,b}}, w_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}, \mathsf{PP})$, where the string $\mathcal{S}$ is set to be $\mathsf{R} \oplus \mathsf{Combiner}$. The output of the hybrid is the output of the adversary.

The computational indistinguishability of $\mathsf{Hybrid}_{2,b}$ and $\mathsf{Hybrid}_{3,b}$ can be argued by introducing an intermediate hybrid $\mathsf{Hybrid}_{2.5,b}$. The indistinguishability of Hybrid_2 and $\mathsf{Hybrid}_{2.5,b}$ as well as the indistinguishability of $\mathsf{Hybrid}_{2.5,b}$ and $\mathsf{Hybrid}_{3,b}$ follows from the $\mathsf{VK}_{\mathsf{one}}$-indistinguishability property of the underlying splittable signatures scheme. This is because, at any time, the adversary is handed over only signatures on $(\mathsf{acc}_{x_{0,b}}, 0)$ and $(\mathsf{acc}_{x_{1,b}}, 1)$. Thus, the verification key VK_x (or VK_y) can be replaced with $\mathsf{VK}_{\mathsf{one},0}$ (or $\mathsf{VK}_{\mathsf{one},1}$) without a computationally bounded adversary being able to distinguish this change with significant probability.

$\underline{\mathsf{Hybrid}_{4,b} \text{ for } b \in \{0,1\}}$: In this hybrid, the challenger hardwires the signature of the "correct" combined accumulator value in the combiner algorithm.

The challenger produces the secret parameters $(\mathsf{3StgKLW.sk}, \mathsf{R}, \{\mathsf{SK}_c, \mathsf{VK}_c\}_{c \in \{0,1\}})$ by executing $\mathsf{SRE.Gen}(1^\lambda, T)$. If the adversary submits query for the input $((x_{0,0}, x_{0,1}), 0)$ of the form first, the challenger responds by computing an input encoding of $x_{0,b}$, namely $(\widehat{x_{0,b}}, \mathsf{acc}_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}_0)$, computed as in the scheme, except that $\mathcal{S}_0$ is just set to be R, which is part of the secret parameters output by the SRE setup algorithm. Next, when the adversary submits the query $((x_{1,0}, x_{1,1}), 1)$ to the challenger. The challenger then responds with an input encoding of $x_{1,b}$, namely $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_1)$, which are computed as in the scheme, except that $\mathcal{S}_1$ in the encoding is set to be $\mathsf{R} \oplus \mathsf{Combiner}$, where $\mathsf{Combiner}$ is obtained by obfuscating the program $\mathsf{Prog2}\{\mathsf{VK}_{\mathsf{one},0}, \mathsf{VK}_{\mathsf{one},1}, \perp, \mathsf{acc}_{x_{1,0}||x_1}\}$, as described in Figure 2. Here, $\mathsf{VK}_{\mathsf{one},0}$ and $\mathsf{VK}_{\mathsf{one},1}$ are generated as in the previous hybrid and the value $\mathsf{acc}_{x||y}$ is obtained by executing the 3-stage KLW combine algorithm $\mathsf{3StgKLW.Combine}$ on input $(\mathsf{3StgKLW.sk}, \mathsf{acc}_{x_{0,b}}, \mathsf{acc}_{x_{1,b}})$.

[6]Here, since the splittable signature scheme is deterministic, there is only such signature

Similarly, if the adversary queries for $((x_{1,0}, x_{1,1}), 1)$ then the challenger responds with $(\widehat{x_{1,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}}, \mathcal{S}_0)$, where it sets $\mathcal{S}_0$ to be R and when the adversary queries for $((x_{0,0}, x_{0,1}), 0)$, the challenger responds with $(\mathsf{CT}_{x_{0,b}}, w_{x_{0,b}}, \sigma_{x_{0,b}}, \mathcal{S}, \mathsf{PP})$, where the string $\mathcal{S}$ is set to be $\mathsf{R} \oplus \mathsf{Combiner}$. The output of the hybrid is the output of the adversary.

The indistinguishability of $\mathsf{Hybrid}_{3,b}$ and $\mathsf{Hybrid}_{4,b}$ follows from the security of indistinguishability obfuscation. The reason is that the programs $\mathsf{Prog2}\{\mathsf{VK}_{\mathsf{one},0}, \mathsf{VK}_{\mathsf{one},1}, \bot, \mathsf{acc}_{x_{1,0}||x_1}\}$ and $\mathsf{Prog2}\{\mathsf{VK}_{\mathsf{one},0}, \mathsf{VK}_{\mathsf{one},1}, \mathsf{SRE.sk}, \bot\}$ are functionally equivalent. This is because, both the programs output $\bot$ on all the inputs except the input $(\mathsf{acc}_{x_{0,b}}, \sigma_{x_{0,b}}, \mathsf{acc}_{x_{1,b}}, \sigma_{x_{1,b}})$. On this input, the program with $\mathsf{SRE.sk}$ calculates the joint value to be $\mathsf{acc}_{x_{1,0}||x_{1,1}}$ and the program without the secret parameters essentially outputs this value, $\mathsf{acc}_{x_{1,0}||x_{1,1}}$, which has already been hardwired.

Finally, the indistinguishability of $\mathsf{Hybrid}_{4,0}$ and $\mathsf{Hybrid}_{4,1}$ follows from the security of $\mathsf{3StgKLW}$ scheme.

□

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 [GGH+13, Wat15] (here, we refer to the post-challenge FE construction of [Wat15]). 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 [BS14] on the public-key FE [7]. 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, [BS14] does not add any additional assumptions).

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.

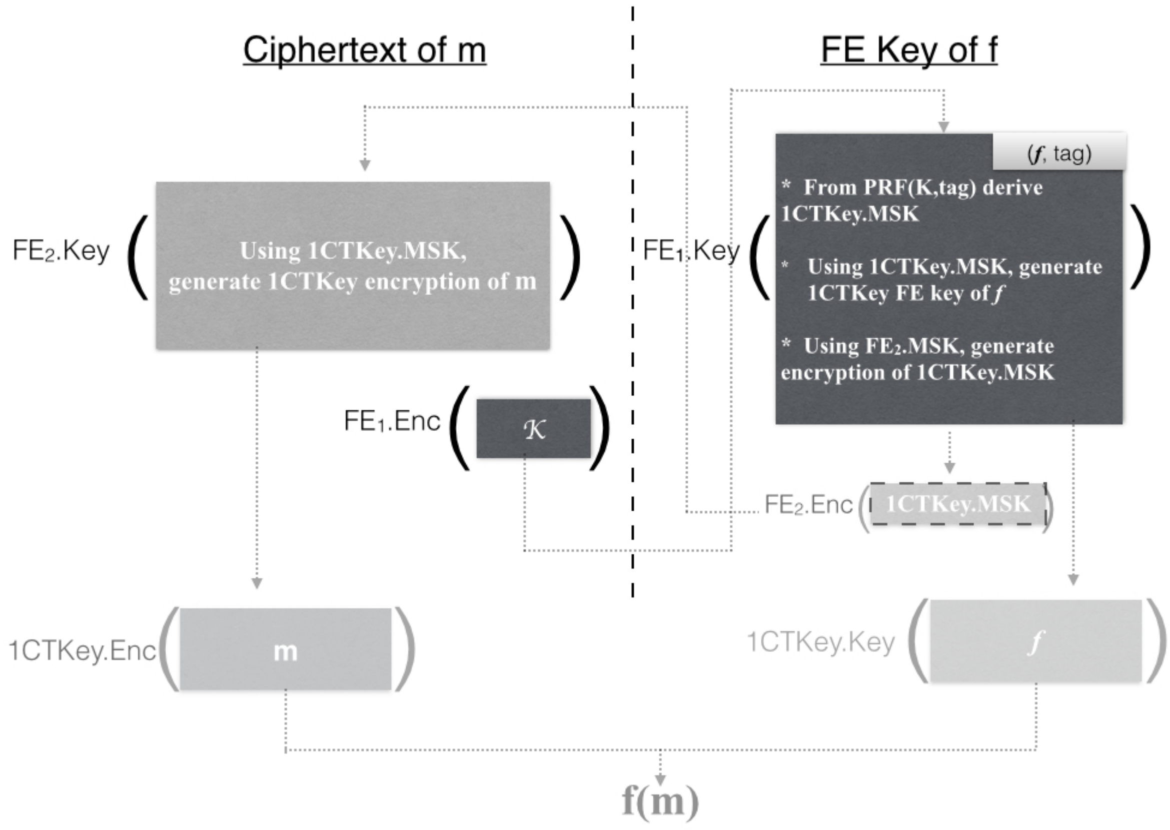
The key idea is as follows: we give a mechanism to generate a *unique key* corresponding to a pair of ciphertext (of m) and functional key (of f) in the resulting adaptive multi-key FE scheme. This unique key would correspond to the master secret key of the adaptive 1-CT 1-Key FE scheme. At this point, 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 [Wat15]. We show how to implement the same, in the more general context

[7]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.

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 3.

Our construction has more details that we present below.



Ciphertext of m
FE Key of f
FE₂.Key
Using 1CTKey.MSK, generate 1CTKey encryption of m
FE₁.Enc
K
FE₁.Key
(f, tag)
* From PRF(K,tag) derive 1CTKey.MSK
* Using 1CTKey.MSK, generate 1CTKey FE key of f
* Using FE₂.MSK, generate encryption of 1CTKey.MSK
FE₂.Enc
1CTKey.MSK
1CTKey.Enc
m
1CTKey.Key
f
f(m)

Figure 3 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.

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 $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.

$\underline{\mathsf{KeyGen}(\mathsf{MSK} = \mathsf{PubFE.MSK}, f)}$: Draw C_E at random[8]. Denote τ to be $(\tau_0||\tau_1||\tau_2||\tau_3)$, where τ_i for $i \in \{0,1,2,3\}$ is picked at random. Execute $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f, C_E, \tau])$, where $G[f, C_E, \tau]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$. Output $sk_f = \mathsf{PubFE}.sk_G$.

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

1. Parse $\tau$ 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 $(\mathsf{OneCTKey}.sk_f, \mathsf{OneKey.CT})$
```

Figure 4

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

- 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 5.
- Execute $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\mathsf{OneKey.MSK}, \mathsf{K}, \perp, 0))$ to obtain $\mathsf{PubFE.CT}$.

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

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

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

Figure 5

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

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

[8]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 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}.

Output $\hat{m}$.

We argue in Section 4.1 that the above scheme is correct, succinct and has input-specific run time property. We then prove the following theorem that establishes the proof of security of the above scheme.

Theorem 5. *Assuming the selective security of* $\mathsf{PubFE}, \mathsf{OneKey}$*, adaptive security of* $\mathsf{OneCTKey}$*, 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 prove the security formally in the Section 6.

4.1 Proof of Correctness and Efficiency

Correctness. To argue correctness, we need to show that the output of the decryption algorithm on input a functional key of f and an encryption of m is $f(m)$. Recall that the functional key of f is nothing but a PubFE functional key of G and the ciphertext of m consists of a OneKey functional key of $H[m]$ and a PubFE encryption of $(\mathsf{OneKey.MSK}, \mathsf{K}, \perp, 0)$ (where the notation is as described in the construction). From the correctness of PubFE, we have that the output of $\mathsf{PubFE.Dec}$ on input a functional key of G and an encryption of $(\mathsf{OneKey.MSK}, \mathsf{K}, \perp, 0)$ is a $\mathsf{OneCTKey}$ functional key of f and a $\mathsf{OneKey.CT}$ encryption of $(\mathsf{OneCTKey.MSK}, R_2, 0)$ (with notation as in the construction). Further, from the correctness of OneKey, we have the output of $\mathsf{OneKey.Dec}$ on input, a functional key of $H[m]$ and encryption of $(\mathsf{OneCTKey.MSK}, R_2, 0)$, is a $\mathsf{OneCTKey}$ encryption of m. Finally, the correctness of $\mathsf{OneCTKey}$ ensures that the output of $\mathsf{OneCTKey.Dec}$ on input a functional key of f and an encryption of m is $f(m)$, as required.

Efficiency. We argue, one by one, that all the algorithms in the above scheme satisfies the efficiency conditions.

1. The running time of $\mathsf{Setup}(1^\lambda)$ is the same as the running time of $\mathsf{PubFE.Setup}(1^\lambda)$, which is a polynomial in λ.
2. The running time of $\mathsf{KeyGen}(1^\lambda, |f|)$ is equal to a polynomial in t, where t is equal to the running time of $\mathsf{PubFE.KeyGen}$ on input $(\mathsf{PubFE.MSK}, G[f, C_E, \tau])$, which is again a polynomial in $(\lambda, |G[f, C_E, \tau]|)$ (note that G is represented as a circuit). We calculate $|G[f, C_E, \tau]|$ as follows. First, note that the size of $\mathsf{OneCTKey.Setup}$ (as a circuit) is a polynomial in λ. The size of $\mathsf{OneCTKey.KeyGen}$ is polynomial in $(\lambda, |f|)$. Then, the size of $\mathsf{OneKey.Enc}$ is polynomial in λ. Lastly, the size of $\mathsf{Sym.Dec}$ is again a polynomial in $(\lambda, |C_E|)$, where $|C_E|$ is a polynomial in $(\lambda, |f|)$. Using these facts, we have that $|G[f, C_E, \tau]|$ is a polynomial in $(\lambda, |f|)$ and thus, the running time of KeyGen is a polynomial in $(\lambda, |f|)$.
3. We calculate the running time of $\mathsf{Enc}(\mathsf{PK}, m)$ step-by-step. Sampling of the PRF key K takes time polynomial in λ. In the second step, the time taken to execute $\mathsf{OneKey.Setup}$ is again a polynomial in λ. In the third step, the running time of $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}, H[m])$ is a polynomial in $(\lambda, |H[m]|)$. The size of $|H[m]|$ (same as $|H^*[m, m', v]|$, as in Figure 7) is a a polynomial in $(\lambda, |\mathsf{OneCTKey.Enc})|)$, where the size of $\mathsf{OneCTKey.Enc}$ is again a polynomial in $(\lambda, |m|)$. Thus, the running time of $\mathsf{OneKey.KeyGen}$ is a polynomial in $(\lambda, |m|)$. In the final step of Enc, the $\mathsf{PubFE.Enc}$ algorithm takes time polynomial in λ. Summing up the running time of all the steps, we get the running time of Enc to be poly in $(\lambda, |m|)$ as intended.
4. Finally, we determine the running time of $\mathsf{Dec}(\mathsf{OneKey}.sk_H, \mathsf{PubFE.CT})$. In the first step, the running time of $\mathsf{PubFE.Dec}$ is a polynomial in $(\lambda, |M|, |G|)$, where $M = (\mathsf{OneKey.MSK}, \mathsf{K}, \perp, 0)$ [9]. We have already seen that $|G|$ is a polynomial in $(\lambda, |f|)$ and further, observe that $|M|$ is a polynomial in λ. In the second step, the running time of $\mathsf{OneKey.Dec}$ is a polynomial in $(\lambda, |H|, |M'|)$, where $M' = (\mathsf{OneCTKey.MSK}, R_2, 0)$. We have already seen that H is polynomial in $(\lambda, |m|)$ and further, $|M'|$ is a polynomial in λ. Lastly, the running time of $\mathsf{OneCTKey.Dec}$ is a polynomial in $(\lambda, |f|, |m|, \mathsf{timeTM}(f, m))$. Thus, the running time of Dec is a polynomial in $(\lambda, |f|, |m|, \mathsf{timeTM}(f, m))$, as desired.

[9]Note that the $\perp$ symbol is viewed as a string of size $|\mathsf{Sym}.k|$, which is a polynomial in λ.

5 Proof of Theorem 5: Informal

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 $1/2 + \mathsf{negl}(\lambda)$.

$\underline{\mathsf{Hybrid}_0}$: 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.

$\underline{\mathsf{Hybrid}_1}$: 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.

$\underline{\mathsf{Hybrid}_2}$: 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 $(\perp, \perp, \mathsf{Sym}.k^*, 1)$ instead of $(\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0)$, 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 4) on input $(\perp, \perp, \mathsf{Sym}.k^*, 1)$ 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 $(\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0)$.

$\underline{\mathsf{Hybrid}_3}$: 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 6. 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 $(\mathsf{OneCTKey.MSK}, R, 0)$.

$\underline{\mathsf{Hybrid}_5}$: We switch the mode of α from 0 to 1 in the OneKey ciphertexts output by all the functional keys.

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

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 $v$.
```

Figure 6

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 Proof of Theorem 5: Formal

As remarked earlier, we present the proof of *selective* security of FE. We point out the (minor) changes in the proof that need to be made in order to prove the *adaptive* security of FE.

We prove the theorem by giving a sequence of hybrids. The first hybrid corresponds to the real experiment, where the challenger uses a bit b picked at random, to generate the challenge ciphertext as in the 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 b is $1/2$. And thus the probability that the adversary guesses b correctly in the first hybrid is at most $1/2 + \mathsf{negl}(\lambda)$.

We define the advantage of an adversary $\mathcal{A}$ in hybrid, say Hybrid_i, denoted by $\mathsf{Adv}^i_{\mathcal{A}}$, to be the probability that $\mathcal{A}$ outputs b.

$\underline{\mathsf{Hybrid}_0}$**:** 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.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.

3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
5. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H[m_b])$ to obtain $\mathsf{OneKey}.sk^*_H$, where H is defined in Figure 5.
6. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_H, \mathsf{PubFE.CT}^*)$.
7. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
8. For the j^{th} function query f_j made by the adversary, the challenger does the following.
   (a) Challenger draws $C_E, \tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
   (b) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j])$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
   (c) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
9. Adversary outputs bit b' and wins if $b' = b$.

Hybrid$_1$: In this hybrid, the values corresponding to the challenge ciphertext are hardwired in the "C_E" component of all the functional keys.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H[m_b])$ to obtain $\mathsf{OneKey}.sk^*_H$, where H is defined in Figure 5.
7. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_H, \mathsf{PubFE.CT}^*)$.
8. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
9. For the j^{th} function query f_j made by the adversary, the challenger does the following.
   (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
   (b) Challenger computes $R_i \leftarrow \mathsf{PRF}(\mathsf{K}^*, \tau_i)$, for $i \in \{0, 1, 2, 3\}$. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R_1)$. It computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R_2, 0); R_3)$. It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
   (c) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
   (d) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
10. Adversary outputs bit b' and wins if $b' = b$.

Lemma 2. *Assuming the security of symmetric encryption scheme* Sym, *for any PPT adversary $\mathcal{A}$, we have* $|\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^0_{\mathcal{A}}| \le \mathsf{negl}(\lambda)$.

$\underline{\textsf{Hybrid}_2}$: In this hybrid, switch the mode from $\beta = 0$ to $\beta = 1$ in the challenge ciphertext. In the challenge ciphertext, the $(\textsf{PubFE})$ encryption of $(\perp, \perp, \textsf{Sym}.k^*, 1)$ is computed instead of $(\textsf{OneKey.MSK}^*, \textsf{K}^*, \perp, 0)$.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0,1\}$ at random.
3. Challenger executes $\textsf{PubFE.Setup}(1^\lambda)$ to obtain $(\textsf{PubFE.MSK}, \textsf{PubFE.PK})$. It then sets $(\textsf{MSK} = \textsf{PubFE.MSK}, \textsf{PK} = \textsf{PubFE.PK})$.
4. Challenger executes $\textsf{Sym.Setup}(1^\lambda)$ to obtain $\textsf{Sym}.k^*$.
5. Challenger chooses a PRF key $\textsf{K}^*$ at random. Challenger executes $\textsf{OneKey.Setup}(1^\lambda)$ to obtain $\textsf{OneKey.MSK}^*$.
6. Challenger executes $\textsf{OneKey.KeyGen}(\textsf{OneKey.MSK}^*, H[m_b])$ to obtain $\textsf{OneKey}.sk_H^*$, where H is defined in Figure 5.
7. Challenger executes $\underline{\textsf{PubFE.Enc}(\textsf{PubFE.PK}, (\perp, \perp, \textsf{Sym}.k^*, 1))}$ to obtain $\textsf{PubFE.CT}^*$. It then sets $\textsf{CT}^* = (\textsf{OneKey}.sk_H^*, \textsf{PubFE.CT}^*)$.
8. Challenger then sends $(\textsf{PK}, \textsf{CT}^*)$ to the adversary.
9. For the j^{th} function query f_j made by the adversary, the challenger does the following.
   (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
   (b) Challenger computes $R_i \leftarrow \textsf{PRF}(\textsf{K}^*, \tau_i)$, for $i \in \{0,1,2,3\}$. It computes $\textsf{OneCTKey.MSK}$ to be the output of $\textsf{OneCTKey.Setup}(1^\lambda; R_0)$. It then computes $\textsf{OneCTKey}.sk_f$ to be the output of $\textsf{OneCTKey.KeyGen}(\textsf{OneCTKey.MSK}, f; R_1)$. It computes $\textsf{OneKey.CT}$ to be the output of $\textsf{OneKey.Enc}(\textsf{OneKey.MSK}^*, (\textsf{OneCTKey.MSK}, R_2, 0); R_3)$. It then computes C_E to be the output of $\textsf{Sym.Enc}(\textsf{Sym}.k^*, u)$, where $u = (\textsf{OneCTKey}.sk_{f_j}, \textsf{OneKey.CT})$.
   (c) Challenger executes $\textsf{PubFE.KeyGen}(\textsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\textsf{PubFE}.sk_G$.
   (d) Challenger sends $sk_{f_j} = \textsf{PubFE}.sk_G$ to the adversary.
10. Adversary outputs bit b' and wins if $b' = b$.

Lemma 3. *Assuming the security of public-key FE scheme* $\textsf{PubFE}$*, for any PPT adversary* $\mathcal{A}$*, we have* $|\textsf{Adv}^2_{\mathcal{A}} - \textsf{Adv}^1_{\mathcal{A}}| \leq \textsf{negl}(\lambda)$.

$\underline{\textsf{Hybrid}_3}$: The hardwired values in the "C_E" components of all the functional keys are now computed using randomness drawn from a uniform distribution, as against computing them using a PRF.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0,1\}$ at random.
3. Challenger executes $\textsf{PubFE.Setup}(1^\lambda)$ to obtain $(\textsf{PubFE.MSK}, \textsf{PubFE.PK})$. It then sets $(\textsf{MSK} = \textsf{PubFE.MSK}, \textsf{PK} = \textsf{PubFE.PK})$.
4. Challenger executes $\textsf{Sym.Setup}(1^\lambda)$ to obtain $\textsf{Sym}.k^*$.
5. Challenger chooses a PRF key $\textsf{K}^*$ at random. Challenger executes $\textsf{OneKey.Setup}(1^\lambda)$ to obtain $\textsf{OneKey.MSK}^*$.
6. Challenger executes $\textsf{OneKey.KeyGen}(\textsf{OneKey.MSK}^*, H[m_b])$ to obtain $\textsf{OneKey}.sk_H^*$, where H is defined in Figure 5.
7. Challenger executes $\textsf{PubFE.Enc}(\textsf{PubFE.PK}, (\perp, \perp, \textsf{Sym}.k^*, 1))$ to obtain $\textsf{PubFE.CT}^*$. It then sets $\textsf{CT}^* = (\textsf{OneKey}.sk_H^*, \textsf{PubFE.CT}^*)$.
8. Challenger then sends $(\textsf{PK}, \textsf{CT}^*)$ to the adversary.
9. For the j^{th} function query f_j made by the adversary, the challenger does the following.
   (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
   (b) $\underline{\text{Challenger computes } R_i^*\text{, for } i \in \{0,1,2,3\}\text{, at random}}$. It computes $\textsf{OneCTKey.MSK}$ to be the output of $\textsf{OneCTKey.Setup}(1^\lambda; R_0^*)$. It then computes $\textsf{OneCTKey}.sk_f$ to be the output of $\textsf{OneCTKey.KeyGen}(\textsf{OneCTKey.MSK}, f; R_1^*)$. It computes $\textsf{OneKey.CT}$ to be the output of $\textsf{OneKey.Enc}(\textsf{OneKey.MSK}^*, (\textsf{OneCTKey.MSK}, R_2^*, 0); R_3^*)$. It then computes C_E to be the output of $\textsf{Sym.Enc}(\textsf{Sym}.k^*, u)$, where $u = (\textsf{OneCTKey}.sk_{f_j}, \textsf{OneKey.CT})$.

(c) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.

(d) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.

10. Adversary outputs bit b' and wins if $b' = b$.

Lemma 4. *Assuming the security of pseudorandom function family* F*, for any PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^2_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

$\underline{\mathsf{Hybrid}_4}$**:** A branch encrypting message m_0 (the 0^{th} message in the challenge message query) is introduced in the function H.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is a random string[10], to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
7. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
8. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
9. For the j^{th} function query f_j made by the adversary, the challenger does the following.
   (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0 || \tau_1 || \tau_2 || \tau_3)$.
   (b) Challenger computes R^*_i, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R^*_1)$. It computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 0); R^*_3)$. It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
   (c) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
   (d) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
10. Adversary outputs bit b' and wins if $b' = b$.

Lemma 5. *Assuming the selective security of function-private single-key FE scheme* OneKey*, for any PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^3_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

$\underline{\mathsf{Hybrid}_5}$**:** We switch the mode of α from 0 to 1 in the OneKey ciphertexts output by all the functional keys.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is a random string, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.

[10]The length of v is determined to be the length of a $\mathsf{OneCTKey}$ ciphertext of a message of length $|m_0|$ (or $|m_b|$).

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

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 $v$.
```

Figure 7

7. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
8. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
9. For the j^{th} function query f_j made by the adversary, the challenger does the following.
    (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
    (b) Challenger computes R^*_i, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R^*_1)$. It computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 1); R^*_3)$. It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
    (c) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
    (d) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
10. Adversary outputs bit b' and wins if $b' = b$.

Lemma 6. *Assuming the security of function-private single-key FE scheme* OneKey*, pseudorandom function family* F *and single-key single-ciphertext FE* OneCTKey*, for any PPT adversary $\mathcal{A}$, we have* $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

$\underline{\mathsf{Hybrid}_6}$**:** We change the $\alpha = 0$ branch in the function H to encrypt the message m_0 instead of m_b.

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger computes R_i, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}_i$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R^*_1)$. It computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 1); R^*_3)$.
7. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_0, m_0, v])$, where v is a random string, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
8. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.

9. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
10. For the j^{th} function query f_j made by the adversary, the challenger does the following.
    (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
    (b) Challenger computes R_i^*, for $i \in \{0,1,2,3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R_0^*)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK},\ f; R_1^*)$. It computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R_2^*, 1); R_3^*)$. It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
    (c) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
    (d) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
11. Adversary outputs bit b' and wins if $b' = b$.

Lemma 7. *Assuming the selective security of function-private single-key FE scheme, for any PPT adversary $\mathcal{A}$, we have $|\mathsf{Adv}^6_{\mathcal{A}} - \mathsf{Adv}^5_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.*

Lemma 8. *For any PPT adversary $\mathcal{A}$, we have $\mathsf{Adv}^6_{\mathcal{A}} = \frac{1}{2}$.*

Proof. Notice that in Hybrid_6, that the bit b picked by the challenger is completely hidden from the adversary and more specifically, none of the messages exchanged between the challenger and the adversary has any information about b. Hence, the probability that the adversary guesses b' such that $b' = b$ is exactly 1/2. □

From Lemmas 2, 3, 4, 5, 6 and 7, we have that

$$|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}| \leq \mathsf{negl}(\lambda). \tag{1}$$

Recall that $\mathsf{Adv}^6_{\mathcal{A}} = 1/2$, from Lemma 8. Plugging this in Equation 1, we have that $\mathsf{Adv}^0_{\mathcal{A}}$ is at most $\frac{1}{2} + \mathsf{negl}(\lambda)$. This completes the proof of the theorem.

6.1 Security Proof: Indistinguishability of Hybrids

Proof of Lemma 2. We design a PPT reduction $\mathcal{B}$ that internally uses $\mathcal{A}$ to break the security of symmetric encryption scheme with probability $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}|$, thus proving the lemma. Here, $\mathcal{B}$ simulates the role of the challenger in the selective security game of FE.

When the adversary $\mathcal{A}$ sends the message pair query (m_0, m_1) to $\mathcal{B}$, it first generates the public key PK and then generates the challenge ciphertext CT^* of m_b with b picked at random. Both PK and CT^* are generated as in Hybrid_0. For every function query f_j, where $j \in [q]$ the reduction $\mathcal{B}$, generates the values $u_j = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$ as in Hybrid_1. It then sends the message u_j to the challenger of the Sym security game. In return it gets back C_E. The reduction $\mathcal{B}$ then uses this to design the circuit $G[f_j, C_E, \tau_j]$ as in Hybrid_1. It then computes a PubFE functional key of $G[f_j, C_E, \tau_j]$ to obtain $\mathsf{PubFE}.sk_G$, which is then sent to the adversary. In the end, $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs.

If the challenger of Sym security game sends a random string (for every query) then we are in Hybrid_0, else if it sends an encryption of u_j, for every $j \in [q]$, then we are in Hybrid_1. Hence, the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}|$, which proves the lemma.

Proof of Lemma 3. We prove by contradiction. Suppose there exists an PPT adversary $\mathcal{A}$ such that the difference in the advantages of $\mathcal{A}$ in Hybrid_1 and Hybrid_2 is non-negligible then we construct a $\mathcal{B}$ that breaks the security of PubFE with non-negligible probability. Our reduction $\mathcal{B}$ will simulate the role of the challenger in the selective security game of FE.

When the adversary $\mathcal{A}$ makes the message pair query (m_0, m_1), then $\mathcal{B}$ does the following. It first generates $\mathsf{OneKey.MSK}^*$ by executing $\mathsf{OneKey.Setup}(1^\lambda)$. It then samples a PRF key K^*. It also computes a symmetric key $\mathsf{Sym}.k$ by executing $\mathsf{Sym.Setup}(1^\lambda)$. It then sends the challenge message query $\Big((\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0), (\perp, \perp, \mathsf{Sym}.k^*)\Big)$ to the challenger of PubFE (selective) security game. In return it receives back $(\mathsf{PubFE.PK}, \mathsf{PubFE.CT}^*)$. The reduction $\mathcal{B}$ then proceeds to generates a $\mathsf{OneKey}.sk^*_H$ functional key of $H[m_b]$ as in Hybrid_2, where b is picked at random. It then sends the public key $\mathsf{PK} = \mathsf{PubFE.CT}^*$ along with the challenge ciphertext $\mathsf{CT}^* = (\mathsf{PubFE.CT}^*, \mathsf{OneKey}.sk_H^*)$ to $\mathcal{A}$. For every function query f_j made by $\mathcal{A}$, the reduction $\mathcal{B}$ constructs the circuit $G[f_j, C_E, \tau_j]$ as in Hybrid_2. It then forwards it to the challenger of PubFE and in return it receives $\mathsf{PubFE}.sk_G$. $\mathcal{B}$ then sets $sk_f = \mathsf{PubFE}.sk_G$ and sends it to $\mathcal{A}$. At the end of the execution, the output of $\mathcal{B}$ is the output of $\mathcal{A}$.

We first prove that $\mathcal{B}$ is a valid adversary of the PubFE game. To see this, notice that for every function query f_j, with $j \in [q]$, we have $G[f_j, C_E, \tau_j](\perp, \perp, \mathsf{Sym}.k^*, 1)$ is the decrypted value of C_E. And this is nothing but the output of $G[f_j, C_E, \tau_j](\mathsf{OneKey.MSK}^*, \mathsf{K}, \perp, 0)$ by the way we defined the message to be encrypted in C_E. Further, the challenge message query is made at the beginning of the game. This shows that $\mathcal{B}$ is a valid adversary.

If the challenger sent back an encryption of $(\mathsf{OneKey.MSK}^*, \mathsf{K}^*, \perp, 0)$ then we are in Hybrid_0 else if it sent back an encryption of $(\perp, \perp, \mathsf{Sym}.k^*, 1)$ then we are in Hybrid_1. Hence, the success probability of $\mathcal{B}$ directly translates to the difference in the advantages of $\mathcal{A}$ in Hybrid_1 and Hybrid_2, which contradicts the security of PubFE. This completes the proof of the lemma.

Remark 4. *Notice that the above proof goes through if we were proving the adaptive security of* FE *while still assuming the selective security of* PubFE.

Proof of Lemma 4. Suppose there exists a PPT adversary $\mathcal{A}$ such that $|\mathsf{Adv}^2_{\mathcal{A}} - \mathsf{Adv}^3_{\mathcal{A}}|$ is non-negligible, call it δ. We then construct a PPT reduction $\mathcal{B}$ that internally uses $\mathcal{A}$ to break the security of pseudorandom functions with $\mathcal{B}$ playing the role of the challenger of FE when interacting with $\mathcal{A}$.

When $\mathcal{A}$ makes the challenge message query, $\mathcal{B}$ computes the public key and the challenge ciphertext as in Hybrid_2 and then sends them to $\mathcal{A}$. For every function query f_j, $\mathcal{B}$ picks τ_i, for $i \in \{0, 1, 2, 3\}$, at random. It then submits queries τ_i, for $i \in \{0, 1, 2, 3\}$ to the PRF oracle it has access to. In return it receives R_i^*, for $i \in \{0, 1, 2, 3\}$. The reduction $\mathcal{B}$ then proceeds as in Hybrid_3 to generate the functional key sk_{f_j}, which it then sends to $\mathcal{A}$. The output of $\mathcal{B}$ is essentially the output of $\mathcal{A}$.

If the values R_i^* (for every function query) were the output of a pseudorandom function then we are in Hybrid_2, else we are in Hybrid_3. From our hypothesis, this means that $\mathcal{B}$ breaks the security of the pseudorandom function family F with non-negligible probability δ; a contradiction. This proves the lemma.

Proof of Lemma 5. We design a reduction $\mathcal{B}$, that uses $\mathcal{A}$, to break the function privacy property of OneKey. We then claim that the success probability of $\mathcal{B}$ is exactly $|\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}}|$ which will prove the lemma. Our reduction $\mathcal{B}$ will simulate the challenger of OneKey to the adversary $\mathcal{A}$.

On receiving the challenge message pair query (m_0, m_1) from $\mathcal{A}$, $\mathcal{B}$ first generates $\mathsf{PK} = \mathsf{PubFE.PK}$ as described in Hybrid_3. It then computes $\mathsf{PubFE.CT}^*$, again as in Hybrid_3. It then computes the message tuple $(x_1, \ldots, x_q)$, where q is the number of function queries[11], as follows. It sets x_j to be $(\mathsf{OneCTKey.MSK}, \mathsf{R}_2^*, 0)$ as in Hybrid_4. The reduction $\mathcal{B}$ then sends the message pair (x_j, x_j), for $j \in [q]$, to the challenger of OneKey. In return it gets back $\mathsf{OneKey.CT}_j$, for $j \in [q]$, which it is going to use later when generating the j^{th} functional key. Once it receives the challenge ciphertexts from the challenger, the reduction $\mathcal{B}$ then constructs two functions $H[m_b]$ (Figure 5) and $H^*[m_b, m_0, v]$ (Figure 7). Now, $\mathcal{B}$ submits the function query $(H[m_b], H^*[m_b, m_0, v])$ to the challenger and then it receives back a functional key $\mathsf{OneKey}.sk_H^*$. The reduction $\mathcal{B}$ then sends $(\mathsf{OneKey}.sk_H^*, \mathsf{PubFE.CT}^*)$ to $\mathcal{A}$. The functional queries made by $\mathcal{A}$ are handled as in Hybrid_4 except that the pre-computed values $\mathsf{OneKey.CT}_j$, for $j \in [q]$ (obtained from the OneKey challenger) are used instead of computing them from scratch. Finally, the output of $\mathcal{B}$ is the same as the output of $\mathcal{A}$.

We show that $\mathcal{B}$ is a valid adversary in the (selective) security game of OneKey. Firstly, $\mathcal{B}$ only makes a single function query to the challenger. Further, it can be seen that $H[m_b](x_j) = H^*[m_b, m_0, v](x_j)$, for every $j \in [q]$ – this is because, x_j is of the form $(*, *, 0)$ and for inputs of that form, the output of $H[m_b](x_j)$ and $H^*[m_b, m_0, v](x_j)$ are the same. Lastly, the challenge message queries are declared by $\mathcal{B}$ at the beginning of the game. This proves that $\mathcal{B}$ is a valid adversary in the security game of OneKey.

If the challenger of OneKey sends a functional key of $H[m_b]$ then we are in Hybrid_3, else if it sends a functional key of $H^*[m_b, m_0, v]$ then we are in Hybrid_4. This proves our claim that the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}}|$.

[11]Here we assume that the reduction knows a priori the number of function queries made by $\mathcal{A}$.

Remark 5. *Notice that if we were proving the adaptive security of* FE*, the above proof goes through while still only assuming the selective security of* OneKey*.*

Proof of Lemma 6. The proof of this lemma involves more intermediate hybrids and hence we defer this proof to the Section 6.1.1.

Proof of Lemma 7. This is similar to the proof of Lemma 5. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}|$ is non-negligible. We then construct a reduction $\mathcal{B}$, that breaks the function privacy of OneKey with probability $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}|$.

The reduction $\mathcal{B}$ on receiving the challenge message pair query (m_0, m_1) from $\mathcal{A}$, first generates $\mathsf{PK} = \mathsf{PubFE.PK}$ as described in Hybrid_5. It then computes $\mathsf{PubFE.CT}^*$, again as in Hybrid_5. It then computes the message tuple $(x_1, \ldots, x_q)$, where q is the number of function queries as follows. It sets x_j to be $(\mathsf{OneCTKey.MSK}, R^*_2, 1)$ as in Hybrid_5. The reduction $\mathcal{B}$ then sends the message pair (x_j, x_j), for $j \in [q]$, to the challenger of OneKey. In return it gets back $\mathsf{OneKey.CT}_j$, for $j \in [q]$, which it is going to use later when generating the j^{th} functional key. Once it receives the challenge ciphertexts from the challenger, the reduction $\mathcal{B}$ then constructs two functions $H^*[m_b, m_0, v]$ and $H^*[m_0, m_0, v]$. Now, $\mathcal{B}$ submits the function query $(H^*[m_b, m_0, v], H^*[m_0, m_0, v])$ to the challenger and then it receives back a functional key $\mathsf{OneKey}.sk^*_H$. The reduction $\mathcal{B}$ then sends $(\mathsf{OneKey}.sk^*_H, \mathsf{PubFE.CT}^*)$ to $\mathcal{A}$. The functional queries made by $\mathcal{A}$ are handled as in Hybrid_4 except that the pre-computed values $\mathsf{OneKey.CT}_j$, for $j \in [q]$ are used instead of computing them from scratch. Finally, the output of $\mathcal{B}$ is the same as the output of $\mathcal{A}$.

As before, we need to show that $\mathcal{B}$ is a valid adversary in the (selective) security game of OneKey. Firstly, $\mathcal{B}$ only makes a single function query to the challenger. Further, it can be seen that $H^*[m_b, m_0, v](x_j) = H^*[m_0, m_0, v](x_j)$, for every $j \in [q]$ – this is because, x_j is of the form $(*, *, 1)$ and for inputs of that form, the output of $H[m_b, m_0, v]$ and $H^*[m_0, m_0, v]$ are the same. Lastly, all the challenge message queries are made at the beginning of the game. This proves that $\mathcal{B}$ is a valid adversary in the security game of OneKey.

If the challenger of OneKey sends a functional key of $H[m_b, m_0, v]$ then we are in Hybrid_3, else if it sends a functional key of $H^*[m_0, m_0, v]$ then we are in Hybrid_4. This proves our claim that the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}|$.

Remark 6. *Notice that if we were proving the adaptive security of* FE*, the above proof goes through while still only assuming the selective security of* OneKey*.*

6.1.1 Proof of Lemma 6

We now describe a sequence of intermediate hybrids, $\mathsf{Hybrid}_{4.\eta.1}, \ldots, \mathsf{Hybrid}_{4,\eta.4}$ for every $\eta \in [q]$, where q is the number of function queries made by the adversary. We also use the notation $\mathsf{Hybrid}_{4.q+1.1}$ to denote Hybrid_5. Our aim is to show the following: for any PPT adversary $\mathcal{A}$, every $\eta \in [q]$,

- $|\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^{4.1.1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.
- $|\mathsf{Adv}^{4.\eta.1}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.2}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.
- $|\mathsf{Adv}^{4.\eta.2}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.3}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.
- $|\mathsf{Adv}^{4.\eta.3}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.
- $|\mathsf{Adv}^{4.\eta.4}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta+1.1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

The above bullets then show that $|\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^5_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$, thus proving Lemma 6.

Before we give the formal hybrids, we first explain the intuition. In both the hybrids Hybrid_4 and Hybrid_5, the function H as part of the challenge ciphertext contains two branches with the first branch encrypting message m_b and the second branch encrypting message m_0. However in Hybrid_4, all the function keys signal the first branch of H while in Hybrid_5, all the function keys signal the second branch of H. So in order to move from Hybrid_4 to Hybrid_5, we create a "dummy" third branch. In order to switch a function key from signaling the first branch to signaling the second branch, we first hardwire the appropriate 1-ciphertext 1-key encryption of m_b in the third branch. At this point, we make the function key to signal the third branch since the first and the third branches are identical. We next switch this encryption from m_b to m_0. We now use the fact that the second and the third branches are identical to switch the function key to signal the second branch instead of the third branch. We repeat this process for every function query.

Hybrid$_{4.\eta.1}$:

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0,1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random and sets $\tau_\eta = \tau_0||\tau_1||\tau_2||\tau_3$. Challenger computes $R^*_{i,\eta}$, for $i \in \{0,1,2,3\}$, at random. It computes $\mathsf{OneCTKey.MSK}_\eta$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_{0,\eta})$. It then computes $\mathsf{OneCTKey}.sk_{f_\eta}$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}_\eta, f_\eta; R^*_{1,\eta})$. It computes $\mathsf{OneKey.CT}_\eta$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}_\eta, R^*_{2,\eta}, 0); R^*_{3,\eta})$. It sets $u_\eta = (\mathsf{OneCTKey}.sk_{f_\eta}, \mathsf{OneKey.CT})$.
7. Challenger then computes v to be the output of $\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}_\eta, m_b; R^*_{2,\eta})$.
8. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is computed as above, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
9. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
10. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
11. For the j^{th} function query f_j made by the adversary, the challenger does the following.
    - If $j \neq \eta$:
      (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
      (b) Challenger computes R^*_i, for $i \in \{0,1,2,3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R^*_1)$.
      (c) If $j < \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, \underline{1}); R^*_3)$. If $j > \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, \underline{0}); R^*_3)$.
      (d) It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
      (e) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
      (f) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
    - If $j = \eta$:
      (a) Challenger computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u_\eta)$. Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_\eta, C_E, \tau_\eta]$, where $G[f_\eta, C_E, \tau_\eta]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
12. Adversary outputs bit b' and wins if $b' = b$.

Lemma 9. *Assuming the selective security of function-private single-key FE scheme* OneKey*, for any PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Adv}^{4.1.1}_{\mathcal{A}} - \mathsf{Adv}^{4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists a PPT adversary $\mathcal{A}$ such that $|\mathsf{Adv}^{4.\eta.1}_{\mathcal{A}} - \mathsf{Adv}^{4}_{\mathcal{A}}|$ is non-negligible. We then design a reduction $\mathcal{B}$ that uses $\mathcal{A}$, by simulating the role of the challenger in to break the function-privacy property of OneKey. We then show that the success probability of $\mathcal{B}$ is $|\mathsf{Adv}^{4.1.1}_{\mathcal{A}} - \mathsf{Adv}^{4}_{\mathcal{A}}|$.

The reduction $\mathcal{B}$ on receiving the challenge message pair query (m_0, m_1) from $\mathcal{A}$, first generates $\mathsf{PK} = \mathsf{PubFE.PK}$ as described in Hybrid_4. It then computes $\mathsf{PubFE.CT}^*$, again as in Hybrid_4. It then computes the message tuple $(x_1, \ldots, x_q)$, where q is the number of function queries as follows. It sets x_j to be $(\mathsf{OneCTKey.MSK}, R^*_2, 0)$ as in Hybrid_4. The reduction $\mathcal{B}$ then sends the message pair (x_j, x_j), for $j \in [q]$, to the challenger of OneKey. In return it gets back $\mathsf{OneKey.CT}_j$, for $j \in [q]$, which it is going to use later when generating the j^{th} functional key. Once it receives the challenge ciphertexts from the challenger, the reduction $\mathcal{B}$ then constructs two functions

$H^*[m_b, m_0, v']$ and $H^*[m_b, m_0, v'']$ (Figure 7), where v' is a random string (of appropriate length) while v'' is computed to be the $\mathsf{OneCTKey}$ encryption of m_0 with respect to $\mathsf{OneCTKey.MSK}_1$ and randomness $R^*_{2,1}$; both of which are associated to the functional query corresponding to 1st function query. $\mathcal{B}$ then submits the function query $(H^*[m_b, m_0, v'], H^*[m_b, m_0, v''])$ to the challenger and then it receives back a functional key $\mathsf{OneKey}.sk^*_H$. The reduction $\mathcal{B}$ then sends $(\mathsf{OneKey}.sk^*_H, \mathsf{PubFE.CT}^*)$ to $\mathcal{A}$.

The functional queries made by $\mathcal{A}$ are handled as in $\mathsf{Hybrid}_{4.1.1}$ except that the pre-computed values $\mathsf{OneKey.CT}_j$, for $j \in [q]$, are used (obtained from the OneKey challenger) instead of computing them from scratch. Finally, the output of $\mathcal{B}$ is the same as the output of $\mathcal{A}$.

We need to show that $\mathcal{B}$ is a valid adversary in the (selective) security game of OneKey. Firstly, $\mathcal{B}$ only makes a single function query to the challenger. Further, it can be seen that $H^*[m_b, m_0, v'](x_j) = H^*[m_b, m_0, v''](x_j)$, for every $j \in [q]$ – this is because, x_j is of the form $(*, *, 0)$ and for inputs of that form, the output of $H[m_b, m_0, v']$ and $H^*[m_b, m_0, v'']$ are the same. Lastly, all the challenge message queries are made at the beginning of the game. This proves that $\mathcal{B}$ is a valid adversary in the security game of OneKey.

If the challenger of OneKey sends a functional key of $H[m_b, m_0, v']$ then we are in Hybrid_4, else if it sends a functional key of $H^*[m_b, m_0, v'']$ then we are in $\mathsf{Hybrid}_{4.1.1}$. This proves our claim that the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}^{4.1.1}_{\mathcal{A}} - \mathsf{Adv}^{4}_{\mathcal{A}}|$. □

Remark 7. *Notice that if we were only proving the adaptive security of* FE*, the above proof goes through even if we were only assuming the selective security of* OneKey*.*

$\underline{\mathsf{Hybrid}_{4.\eta.2}}$**:**

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random and sets $\tau_\eta = \tau_0 || \tau_1 || \tau_2 || \tau_3$. Challenger computes $R^*_{i,\eta}$, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}_\eta$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_{0,\eta})$. It then computes $\mathsf{OneCTKey}.sk_{f_\eta}$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}_\eta, f_\eta; R^*_{1,\eta})$. It computes $\mathsf{OneKey.CT}_\eta$ to be the output of $\underline{\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (0, 0, 2); R^*_{3,\eta})}$. It sets $u_\eta = (\mathsf{OneCTKey}.sk_{f_\eta}, \mathsf{OneKey.CT})$.
7. Challenger then computes v to be the output of $\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}_\eta, m_0; R^*_{2,\eta})$.
8. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is computed as above, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
9. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
10. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
11. For the jth function query f_j made by the adversary, the challenger does the following.
    - If $j \neq \eta$:
      (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0 || \tau_1 || \tau_2 || \tau_3)$.
      (b) Challenger computes R^*_i, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}, f; R^*_1)$.
      (c) If $j < \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 1); R^*_3)$. If $j > \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 0); R^*_3)$.
      (d) It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.

(e) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.

(f) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.

- If $j = \eta$:

(a) Challenger computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u_\eta)$. Challenger executes $\mathsf{PubFE.KeyGen}($ $\mathsf{PubFE.MSK}, G[f_\eta, C_E, \tau_\eta]$, where $G[f_\eta, C_E, \tau_\eta]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.

12. Adversary outputs bit b' and wins if $b' = b$.

Lemma 10. *Assuming the selective security of function-private single-key functional encryption scheme* OneKey*, for any PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Adv}_{\mathcal{A}}^{4.\eta.2} - \mathsf{Adv}_{\mathcal{A}}^{4.\eta.1}| \le \mathsf{negl}(\lambda)$.

Proof. Assume that there exists a PPT adversary $\mathcal{A}$ such that $|\mathsf{Adv}_{\mathcal{A}}^{4.\eta.2} - \mathsf{Adv}_{\mathcal{A}}^{4.\eta.1}|$ is non-negligible. We then construct a reduction $\mathcal{B}$ that breaks the security of the single-key FE scheme. The reduction $\mathcal{B}$ simulates the role of the challenger of OneKey when interacting with $\mathcal{A}$.

The reduction $\mathcal{B}$ on receiving the challenge message pair query (m_0, m_1) from $\mathcal{A}$, first generates $\mathsf{PK} = \mathsf{PubFE.PK}$ after which it computes $\mathsf{PubFE.CT}^*$. These are generated as in $\mathsf{Hybrid}_{4.\eta.1}$. It then computes the message tuple $(x_1, \ldots, x_q)$, where q is the number of function queries as follows. It sets x_j to be $(\mathsf{OneCTKey.MSK}_j, R^*_{2,j}, 1)$, for all $j < \eta$ and it sets x_j to be $(\mathsf{OneCTKey.MSK}_j, R^*_{2,j}, 0)$, for all $j > \eta$, where $\mathsf{OneCTKey.MSK}_j$ is as described in $\mathsf{Hybrid}_{4.\eta.1}$. It then sets x^0_η to be $(\mathsf{OneCTKey.MSK}_\eta, R^*_{2,\eta}, 0)$ and it sets x^1_η to be $(0, 0, 2)$. The reduction $\mathcal{B}$ then sends the message pair (x_j, x_j), for $j \ne \eta$, and (x^0_η, x^1_η) to the challenger of OneKey. In return it gets back $\mathsf{OneKey.CT}_j$, for $j \in [q]$, which it is going to use later when generating the j^{th} functional key. Once it receives the challenge ciphertexts from the challenger, the reduction $\mathcal{B}$ then constructs the function $H^*[m_b, m_0, v]$, which is as computed in $\mathsf{Hybrid}_{4.\eta.1}$. $\mathcal{B}$ then submits the function query $(H^*[m_b, m_0, v], H^*[m_b, m_0, v])$ to the challenger and then it receives back a functional key $\mathsf{OneKey}.sk^*_H$. The reduction $\mathcal{B}$ then sends $(\mathsf{OneKey}.sk^*_H, \mathsf{PubFE.CT}^*)$ to $\mathcal{A}$.

The functional queries made by $\mathcal{A}$ are handled as in $\mathsf{Hybrid}_{4.\eta.1}$. Finally, the output of $\mathcal{B}$ is the same as the output of $\mathcal{A}$.

We need to show that $\mathcal{B}$ is a valid adversary in the (selective) security game of OneKey. Firstly, $\mathcal{B}$ only makes a single function query to the challenger. Further, it can be seen that $H^*[m_b, m_0, v](x^0_\eta) = H^*[m_b, m_0, v](x^1_\eta)$, for every $j \in [q]$. Lastly, the challenge message query is made at the beginning of the game. This proves that OneKey is a valid adversary.

If the challenger of OneKey uses the message x^0_η to compute the η^{th} ciphertext then we are in $\mathsf{Hybrid}_{4.\eta.1}$, else if uses the message x^1_η then we are in $\mathsf{Hybrid}_{4.\eta.2}$. This proves our claim that the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}_{\mathcal{A}}^{4.\eta.2} - \mathsf{Adv}_{\mathcal{A}}^{4.\eta.1}|$. ☐

Remark 8. *Notice that if we were proving the adaptive security of* FE*, the above proof goes through while still only assuming the selective security of* OneKey.

Remark 9. *The hybrids* $\mathsf{Hybrid}_{4.\eta.1}$ *and* $\mathsf{Hybrid}_{4.\eta.2}$ *could be merged into one hybrid, but for simplicity of exposition and to avoid introducing many changes in one hybrid, we choose to not to do so.*

$\underline{\mathsf{Hybrid}_{4.\eta.3}}$**:**

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}$, $\mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.
6. Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random and sets $\tau_\eta = \tau_0 || \tau_1 || \tau_2 || \tau_3$. Challenger computes $R^*_{i,\eta}$, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}_\eta$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_{0,\eta})$. It then computes $\mathsf{OneCTKey}.sk_{f_\eta}$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}_\eta, f_\eta; R^*_{1,\eta})$. It computes $\mathsf{OneKey.CT}_\eta$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (0, 0, 2); R^*_3)$. It sets $u_\eta = (\mathsf{OneCTKey}.sk_{f_\eta}$, $\mathsf{OneKey.CT})$.

7. Challenger then computes v to be the output of $\underline{\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}_\eta, m_0; R^*_{2,\eta})}$.
8. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is computed as above, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
9. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
10. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
11. For the j^{th} function query f_j made by the adversary, the challenger does the following.
    - If $j \neq \eta$:
      (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0 || \tau_1 || \tau_2 || \tau_3)$.
      (b) Challenger computes R^*_i, for $i \in \{0, 1, 2, 3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK},\ f;\ R^*_1)$.
      (c) If $j < \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 1); R^*_3)$. If $j > \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 0); R^*_3)$.
      (d) It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
      (e) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
      (f) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
    - If $j = \eta$:
      (a) Challenger computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u_\eta)$. Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_\eta, C_E, \tau_\eta]$, where $G[f_\eta, C_E, \tau_\eta]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
12. Adversary outputs bit b' and wins if $b' = b$.

Lemma 11. *Assuming the selective security of single-key single-ciphertext FE scheme* OneCTKey*, for any PPT adversary $\mathcal{A}$, we have* $|\mathsf{Adv}^{4.\eta.3}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.2}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. We construct a reduction $\mathcal{B}$ that internally uses $\mathcal{A}$ to break the security of OneCTKey. We further show that the success probability of $\mathcal{B}$ is the same as $|\mathsf{Adv}^{4.\eta.3}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.2}_{\mathcal{A}}|$.

The reduction $\mathcal{B}$ receives the challenge message query (m_0, m_1) from $\mathcal{A}$. It then forwards this to the challenger of OneCTKey selective security game. In return it gets the ciphertext $\mathsf{OneCTKey.CT}^*$ which it denotes it by v. $\mathcal{B}$ then proceeds as in the challenger of $\mathsf{Hybrid}_{4.\eta.3}$ to generate the challenge ciphertext and then to answer function key queries. While generating the challenge ciphertext, $\mathcal{B}$ hardwires v into the function $H[m_b, m_0, v]$.

Now, if the challenger sends an encryption of m_b then we are in $\mathsf{Hybrid}_{4.\eta.2}$ and if the challenger sends an encryption of m_0 then we are in $\mathsf{Hybrid}_{4.\eta.3}$. This proves our claim. □

Remark 10. *Observe that the above proof can be adapted to argue the adaptive security of* FE *by assuming the existence of adaptively secure single-key single-ciphertext FE scheme.*

$\underline{\mathsf{Hybrid}_{4.\eta.4}}$**:**

1. Adversary sends (m_0, m_1) to the challenger.
2. Challenger chooses bit $b \in \{0, 1\}$ at random.
3. Challenger executes $\mathsf{PubFE.Setup}(1^\lambda)$ to obtain $(\mathsf{PubFE.MSK}, \mathsf{PubFE.PK})$. It then sets $(\mathsf{MSK} = \mathsf{PubFE.MSK}, \mathsf{PK} = \mathsf{PubFE.PK})$.
4. Challenger executes $\mathsf{Sym.Setup}(1^\lambda)$ to obtain $\mathsf{Sym}.k^*$.
5. Challenger chooses a PRF key K^* at random. Challenger executes $\mathsf{OneKey.Setup}(1^\lambda)$ to obtain $\mathsf{OneKey.MSK}^*$.

6. Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random and sets $\tau_\eta = \tau_0||\tau_1||\tau_2||\tau_3$. Challenger computes $R^*_{i,\eta}$, for $i \in \{0,1,2,3\}$, at random. It computes $\mathsf{OneCTKey.MSK}_\eta$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_{0,\eta})$. It then computes $\mathsf{OneCTKey}.sk_{f_\eta}$ to be the output of $\mathsf{OneCTKey.KeyGen}(\mathsf{OneCTKey.MSK}_\eta, f_\eta; R^*_{1,\eta})$. It computes $\mathsf{OneKey.CT}_\eta$ to be the output of $\underline{\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}_\eta, R^*_{2,\eta}, 1); R^*_{3,\eta})}$. It sets $u_\eta = (\mathsf{OneCTKey}.sk_{f_\eta}, \mathsf{OneKey.CT})$.
7. Challenger then computes v to be the output of $\mathsf{OneCTKey.Enc}(\mathsf{OneCTKey.MSK}_\eta, m_1; R^*_{2,\eta})$.
8. Challenger executes $\mathsf{OneKey.KeyGen}(\mathsf{OneKey.MSK}^*, H^*[m_b, m_0, v])$, where v is computed as above, to obtain $\mathsf{OneKey}.sk^*_{H^*}$, where H^* is defined in Figure 7.
9. Challenger executes $\mathsf{PubFE.Enc}(\mathsf{PubFE.PK}, (\perp, \perp, \mathsf{Sym}.k^*, 1))$ to obtain $\mathsf{PubFE.CT}^*$. It then sets $\mathsf{CT}^* = (\mathsf{OneKey}.sk^*_{H^*}, \mathsf{PubFE.CT}^*)$.
10. Challenger then sends $(\mathsf{PK}, \mathsf{CT}^*)$ to the adversary.
11. For the j^{th} function query f_j made by the adversary, the challenger does the following.
    - If $j \neq \eta$:
        (a) Challenger draws $\tau_0, \tau_1, \tau_2, \tau_3$ at random. It sets τ_j to be $(\tau_0||\tau_1||\tau_2||\tau_3)$.
        (b) Challenger computes R^*_i, for $i \in \{0,1,2,3\}$, at random. It computes $\mathsf{OneCTKey.MSK}$ to be the output of $\mathsf{OneCTKey.Setup}(1^\lambda; R^*_0)$. It then computes $\mathsf{OneCTKey}.sk_f$ to be the output of $\mathsf{OneCTKey.KeyGen}(\ \mathsf{OneCTKey.MSK},\ f; R^*_1)$.
        (c) If $j < \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 1); R^*_3)$. If $j > \eta$, it computes $\mathsf{OneKey.CT}$ to be the output of $\mathsf{OneKey.Enc}(\mathsf{OneKey.MSK}^*, (\mathsf{OneCTKey.MSK}, R^*_2, 0); R^*_3)$.
        (d) It then computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u)$, where $u = (\mathsf{OneCTKey}.sk_{f_j}, \mathsf{OneKey.CT})$.
        (e) Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_j, C_E, \tau_j]$, where $G[f_j, C_E, \tau_j]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
        (f) Challenger sends $sk_{f_j} = \mathsf{PubFE}.sk_G$ to the adversary.
    - If $j = \eta$:
        (a) Challenger computes C_E to be the output of $\mathsf{Sym.Enc}(\mathsf{Sym}.k^*, u_\eta)$. Challenger executes $\mathsf{PubFE.KeyGen}(\mathsf{PubFE.MSK}, G[f_\eta, C_E, \tau_\eta]$, where $G[f_\eta, C_E, \tau_\eta]$ is described in Figure 4, to obtain $\mathsf{PubFE}.sk_G$.
12. Adversary outputs bit b' and wins if $b' = b$.

Lemma 12. *Assuming the selective security of function-private single-key FE scheme* OneKey, *for any PPT adversary* $\mathcal{A}$, *we have* $|\mathsf{Adv}^{4.\eta.3}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta.4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

This is similar to the proof of the Lemma 9. Here, we will use the indistinguishability of ciphertexts to prove the above lemma.

Lemma 13. *Assuming the selective security of function-private single-key FE scheme* OneKey, *for any PPT adversary* $\mathcal{A}$, *we have* $|\mathsf{Adv}^{4.\eta.4}_{\mathcal{A}} - \mathsf{Adv}^{4.\eta+1.1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

This is similar to the proof of the Lemma 10. Here, we will use indistinguishability of function keys to prove the above lemma.

Acknowledgements

We thank Brent Waters for collaboration at early stages of this project and several discussions. 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.

References

[ABG$^+$13] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. *IACR Cryptology ePrint Archive*, 2013:689, 2013.

[ABSV15] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive functional encryption. In *CRYPTO*, 2015.

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Canetti and Garay [CG13], pages 500–518.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Computationally private randomizing polynomials and their applications. *Computational Complexity*, 15(2):115–162, 2006.

[AIKW15] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate, or how to compress garbled circuit keys. *SIAM Journal on Computing*, 44(2):433–466, 2015.

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *CRYPTO 2015*. Springer, 2015.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In Yehuda Lindell, editor, *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, volume 8349 of *Lecture Notes in Computer Science*, pages 52–73. Springer, 2014.

[BGG$^+$14] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit abe and compact garbled circuits. In *Advances in Cryptology–EUROCRYPT 2014*, pages 533–556. Springer, 2014.

[BGI$^+$12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[BGK$^+$14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In Nguyen and Oswald [NO14], pages 221–238.

[BGL$^+$15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Siddartha Telang. Succinct randomized encodings and their applications. In *STOC*, 2015.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2012.

[BS14] Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. Cryptology ePrint Archive, Report 2014/550, 2014.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography*, pages 253–273. Springer, 2011.

[BSW12] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: a new vision for public-key cryptography. *Commun. ACM*, 55(11):56–64, 2012.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *FOCS*, 2015.

[CCC$^+$15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Computation-trace indistinguishability obfuscation and its applications. *IACR Cryptology ePrint Archive*, 2015:406, 2015.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and ram programs. In *STOC*, 2015.

[CIJ$^+$13] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In Canetti and Garay [CG13], pages 519–535.

[GGG$^+$14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In Nguyen and Oswald [NO14], pages 578–602.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[GGHZ14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Fully secure functional encryption without obfuscation. *IACR Cryptology ePrint Archive*, 2014:666, 2014.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GKP$^+$12] Shafi Goldwasser, Yael Tauman Kalai, Raluca A Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. *IACR Cryptology ePrint Archive*, 2012:733, 2012.

[GKP$^+$13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2009.

[GVW12] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 162–179. Springer, 2012.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, pages 163–172. ACM, 2015.

[IPS15] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In *TCC*, 2015.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.

[KSW08] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, pages 146–162, 2008.

[NO14] Phong Q. Nguyen and Elisabeth Oswald, editors. *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*. Springer, 2014.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[SS10] Amit Sahai and Hakan Seyalioglu. Worry-free encryption: functional encryption with public keys. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 463–472. ACM, 2010.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 457–473, 2005.

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *CRYPTO*, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

A Tools Used in [KLW15]

We recall the tools used in the work of Koppula et al. [KLW15]. The three main tools used in the work of KLW are positional accumulators, iterators and splittable signatures. Positional accumulators can be thought of as a small piece of storage that be used to information-theoretically validate any part of a huge block of memory. An iterator consists of a state that is updated in an iterative fashion as messages are received which is performed using some public parameters. The last tool that we borrow from KLW is the notion of splittable signatures. A splittable signature scheme is just like the standard notion of the signature scheme except that it is equipped with some additional algorithms that produce alternative signing and verification keys with different capabilities. More precisely, there will be "all but one" keys that work correctly except for a single message m^*, and there will be "one" keys that work only for m^*. Furthermore, there also will be a rejection verification key that always output reject no matter what the input signature is.

We now describe the syntax of the tools below.

A.1 Positional Accumulators

The notion of *positional accumulators* is defined below. This is similar in spirit to the notion of statistically binding hash conceived by [HW15]. A positional accumulator for message space Msg_λ consists of the following algorithms.

$\mathsf{SetupAcc}(1^\lambda, T) \rightarrow \mathsf{PP}_{\mathsf{Acc}}, w_0, store_0$ 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 $store_{out}$.

$\mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, w_{in}, m_{write}, \text{INDEX}, aux) \to w_{out}$ **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*.

Correctness. We consider any sequence $(m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)$ of symbols $m_1, \ldots, m_k$ and indices $\text{INDEX}_1, \ldots, \text{INDEX}_k$ each between 0 and $T-1$. We fix any $\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$. For j from 1 to k, we define $store_j$ iteratively as $store_j := \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j)$ and $w_j := Update(\mathsf{PP}_{\mathsf{Acc}}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$. Note that the algorithms other than $\mathsf{SetupAcc}$ are deterministic, so these definitions fix precise values, not random values (conditioned on the fixed starting values $\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0$).

KLW then define various security properties for the above accumulator scheme that – we will not go into its details and we refer the interested reader to their work. A construction of positional accumulators based on iO, puncturable PRFs, and public key encryption was constructed in the same work.

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 $v_0 \in \mathsf{SplScheme}_\lambda$.

$\mathsf{ItrEnforce}(1^\lambda, T, \vec{m} = (m_1, \ldots, m_k))$ 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 $\mathsf{PP}_{\mathsf{Itr}}$ and a state $v_0 \in \mathsf{SplScheme}$.

$\mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{\mathsf{in}}, m)$ The iterate algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Itr}}$, 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 $\mathsf{Iterate}^k(\mathsf{PP}_{\mathsf{Itr}}, v_0, (m_1, \ldots, m_k))$ to denote $\mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{k-1}, m_k)$, where $v_j = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{j-1}, m_j)$ for all $1 \leq j \leq k-1$.
KLW defined the security notions of indistinguishability of Setup and (Information-theoretic) enforcing properties. They further gave a construction of iterators based on indistinguishability obfuscation, puncturable PRFs and the existence of public-key encryption schemes.

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.

Correctness Let $m^* \in \mathsf{Msg}$ be any message. Let $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$ and $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$. Then, we require the following correctness properties:

1. For all $m \in \mathsf{Msg}$, $\mathsf{VerSpl}(\mathsf{VK}, m, \mathsf{SignSpl}(\mathsf{SK}, m)) = 1$.
2. For all $m \in \mathsf{Msg}, m \neq m^*$, $\mathsf{SignSpl}(\mathsf{SK}, m) = \mathsf{SignSplAbo}(\mathsf{SK}_{\mathsf{abo}}, m)$.
3. For all σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{one}}, m^*, \sigma) = \mathsf{VerSpl}(\mathsf{VK}, m^*, \sigma)$.
4. For all $m \neq m^*$ and σ, $\mathsf{VerSpl}(\mathsf{VK}, m, \sigma) = \mathsf{VerSpl}(\mathsf{VK}_{\mathsf{abo}}, m, \sigma)$.
5. For all $m \neq m^*$ and σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{one}}, m, \sigma) = 0$.
6. For all σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{abo}}, m^*, \sigma) = 0$.
7. For all σ and all $m \in \mathsf{Msg}$, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{rej}}, m, \sigma) = 0$.

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.

B Preliminaries (cont'd)

B.1 Standard Notions

Pseudorandom functions. One of the tools that we use in our work are pseudorandom function families. The security property of a pseudorandom function family states that any PPT adversary can distinguish only with negligible probability a function sampled at random from this family from a random function.

Definition 7. *A function family $\mathsf{F} = \left\{\mathsf{PRF}_{\mathsf{K}} : \{0,1\}^n \to \{0,1\}^m : \mathsf{K} \xleftarrow{\$} \mathcal{K}\right\}$, is a* ***pseudorandom function family*** *with $\mathcal{K}$ being the space of PRF keys*[12]*, if the following holds for every security parameter $\lambda \in \mathbb{N}$, every PPT adversary $\mathcal{A}$:*

$$\left|\mathsf{Prob}\left[1 \leftarrow \mathcal{A}^{\mathcal{O}_{(0)}(\mathsf{PRF}_{\mathsf{K}}(\cdot))}(1^\lambda) \ : \ \mathsf{K} \xleftarrow{\$} \mathcal{K}\right] - \mathsf{Prob}\left[1 \leftarrow \mathcal{A}^{\mathcal{O}_{(1)}(\cdot)}(1^\lambda)\right]\right| \leq \mathsf{negl}(\lambda),$$

where the oracle $\mathcal{O}_{(b)}$ takes as input x and outputs $\mathsf{PRF}_{\mathsf{K}}(x)$ if $b = 0$ else if $b = 1$ it outputs r, where r is picked at random from $\{0,1\}^m$.

Pseudorandom functions can be constructed from one-way functions [GGM86, HILL99].

We say that the pseudorandom function family F, defined above, is implementable in NC^1 if every function in F can be implemented by a circuit of depth $c \cdot \log(n)$, for some constant c. We also consider a weakening of the above security notion where the oracle is queried at random points instead of the points being adversarially chosen as defined above. This security notion is termed as *weak* pseudorandom functions.

Symmetric encryption schemes with pseudorandom ciphertexts. Another tool that we use in our transformation is a symmetric encryption scheme. A symmetric encryption scheme consists of a tuple of PPT algorithms $(\mathsf{Sym.Setup}, \mathsf{Sym.Enc}, \mathsf{Sym.Dec})$. The algorithm $\mathsf{Sym.Setup}$, takes as input a security parameter λ in unary and outputs a key K_E. The encryption algorithm takes as input (K_E, m), where K_E is the symmetric key, m is the message to be encrypted and the output is a ciphertext CT. The decryption algorithm takes as input (K_E, CT), where CT is the ciphertext and the output is the message m, contained in the ciphertext.

For this work, we require a symmetric encryption scheme where ciphertexts produced by $\mathsf{Sym.Enc}$ are pseudorandom strings. More formally, we design a game, parametrized by a PPT adversary that has access to an

[12]We abuse the notation and interchangeably denote the PRF keyspace to be a set as well as a sampling procedure that produces PRF keys from this set.

encryption oracle, $\mathcal{O}_b\big(\mathsf{Sym.Enc}(\mathsf{Sym}.k, \cdot)\big)$, parameterized by bit b. The oracle takes as input a message m and outputs $\mathsf{CT} = \mathsf{Sym.Enc}(\mathsf{Sym}.k, m)$ if $b = 0$, else if $b = 1$ it returns a random string $s \in \{0,1\}^{\ell}$, where $\ell = |\mathsf{CT}|$. The adversary wins in this game if the output of adversary is $b' = b$. The success probability of the adversary is defined to be $|\mathsf{Prob}[b' = b] - \frac{1}{2}|$. We say that the scheme is secure if the success probability of the adversary is negligible.

We note that such a symmetric encryption scheme can be constructed from one-way functions, e.g. using weak pseudorandom functions by defining $\mathsf{Sym.Enc}(\mathsf{K}, m; r) = (r, \mathsf{PRF}_{\mathsf{K}}(r) \oplus m)$, see [Gol09] for details.

B.2 Turing machines

We now briefly recall the definition of Turing machines[13]. A Turing machine is a 7-tuple $M = \langle Q, \Sigma_{\text{tape}}, \Sigma_{\text{inp}}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}} \rangle$ where Q and Σ_{tape} are finite sets with the following properties:

- Q is the set of finite states.
- Σ_{inp} is the set of input symbols.
- Σ_{tape} is the set of tape symbols. We will assume $\Sigma_{\text{inp}} \subset \Sigma_{\text{tape}}$ and there is a special blank symbol '$\text{\textvisiblespace}$' $\in \Sigma_{\text{tape}} \setminus \Sigma_{\text{inp}}$.
- $\delta : Q \times \Sigma_{\text{tape}} \to Q \times \Sigma_{\text{tape}} \times \{+1, -1\}$ is the transition function.
- $q_0 \in Q$ is the start state.
- $q_{\text{acc}} \in Q$ is the accept state.
- $q_{\text{rej}} \in Q$ is the reject state, where $q_{\text{acc}} \neq q_{\text{rej}}$.

For any $i \leq T$, we define the following variables:

- $M^T_{\text{tape},i} \in \Sigma^T_{\text{tape}}$: A T dimensional vector which gives the description of tape before i^{th} step.
- $M^T_{\text{pos},i}$: An integer which describes the position of Turing machine header before i^{th} step.
- $M^T_{\text{state},i} \in Q$: This denotes the state of the Turing machine before step i.

Initially, $M^T_{\text{tape},1} = (\text{\textvisiblespace})^T$, $M^T_{\text{pos},1} = 0$ and $M^T_{\text{state},1} = q_0$. At each time step, the Turing machine reads the tape at the head position and based on the current state, computes what needs to be written on the tape, the next state and whether the header must move left or right. More formally, let $(q, \alpha, \beta) = \delta(M^T_{\text{state},i}, M^T_{\text{tape},i}[M^T_{\text{pos},i}])$. Then, $M^T_{\text{tape},i+1}[M^T_{\text{pos},i}] = \alpha$, $M^T_{\text{pos},i+1} = M^T_{\text{pos},i} + \beta$ and $M^T_{\text{state},i+1} = q$. M accepts at time t if $M^T_{\text{state},t+1} = q_{\text{acc}}$. Given any Turing machine M and time bound T, let $\Pi^T_M = 1$ if M accepts within T steps, else $\Pi^T_M = 0$.

Notation. We denote by $\mathsf{timeTM}(M, x)$, the time taken by a Turing machine M to evaluate on input x. We adopt the convention that the Turing machine also outputs, in addition, the time taken to execute. Thus, if we have two inputs x and y, a Turing machine M, then if $M(x) = M(y)$, by this notation, means that not only does M on x output the same value as M on y but also that the running time of M on both x and y are the same.

B.3 Private Key FE for TMs

We can define the notion of private key FE for Turing machines analogous to the public key setting.

A private key functional encryption (FE) scheme PubFE, defined for a class of Turing machines $\mathcal{F}$ and message space $\mathcal{M}$, is represented by four PPT algorithms, namely ($\mathsf{PrivFE.Setup}$, $\mathsf{PrivFE.KeyGen}$, $\mathsf{PrivFE.Enc}$, $\mathsf{PrivFE.Dec}$) whose descriptions are given below.

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

[13]This part is taken verbatim from [KLW15]

As in the public key setting, we can analogously define the efficiency property, in addition to the correctness and the security property. We define the properties 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 $\Pr[f(m) \leftarrow \mathsf{PrivFE.Dec}(\mathsf{PrivFE.KeyGen}(\mathsf{PrivFE.MSK}, f),\ \mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m))] \geq 1 - \mathsf{negl}(\lambda)$, where $\mathsf{PrivFE.MSK} \leftarrow \mathsf{PrivFE.Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Efficiency. The efficiency property of a private-key FE scheme says that the algorithm $\mathsf{PrivFE.Setup}(1^\lambda)$ should run in time polynomial in λ, $\mathsf{PrivFE.KeyGen}$ on input the Turing machine f (along with master secret key) should run in time polynomial in $(\lambda, |f|)$, $\mathsf{PrivFE.Enc}$ on input a message m (along with the master secret key) should run in time polynomial in $(\lambda, |m|)$. Finally, $\mathsf{PrivFE.Dec}$ 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. We give the security definition below.

Definition 8. *A private-key functional encryption scheme* $\mathsf{PubFE} = (\mathsf{PrivFE.Setup}, \mathsf{PrivFE.KeyGen}, \mathsf{PrivFE.Enc}, \mathsf{PrivFE.Dec})$ *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*

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

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

1. *The challenger first executes* $\mathsf{PrivFE.Setup}(1^\lambda)$ *to obtain* $\mathsf{PrivFE.MSK}$. *The adversary makes the following message queries and function queries in no arbitrary order.*
   - ***Message queries**: The adversary submits a message-pair (m_0, m_1) to the challenger. The challenger generates* $\mathsf{CT} = \mathsf{PrivFE.Enc}(\mathsf{PrivFE.MSK}, m_b)$. *If $f(m_0) = f(m_1)$ for all Turing machine queries f made so far, the challenger sends* CT *to the adversary. Otherwise, it aborts.*
   - ***Turing machine queries**: The adversary submits a Turing machine query f to the challenger. The challenger generates* $\mathsf{PrivFE}.sk_f$, *where* $\mathsf{PrivFE}.sk_f$ *is the output of* $\mathsf{PrivFE.KeyGen}(\mathsf{PrivFE.MSK}, f)$. *If $f(m_0) = f(m_1)$, for all message-pair queries (m_0, m_1) made so far, the challenger sends* $\mathsf{PrivFE}.sk_f$ *to the adversary. Otherwise, it aborts.*
2. *The output of the experiment is b', where b' is the output of $\mathcal{A}$.*

We define the scheme to be *selectively secure* if the adversary has to declare all the challenge message pairs at the beginning of the game.