It Runs and it Hides: A Function-Hiding Construction for Private-Key Multi-Input Functional Encryption

Alexandros Bakas[1,3] and Antonis Michalas[1,2]

[1] Tampere University, Tampere, Finland
{alexandros.bakas,antonios.michalas}@tuni.fi
[2] RISE Research Institutes of Sweden, Stockholm, Sweden
[3] Nokia-Bell Labs, Espoo, Finland

Abstract. Functional Encryption (FE) is a modern cryptographic technique that allows users to learn only a specific function of the encrypted data and nothing else about its actual content. While the first notions of security in FE revolved around the privacy of the encrypted data, more recent approaches also consider the privacy of the computed function. While in the public key setting, only a limited level of function-privacy can be achieved, in the private-key setting privacy potential is significantly larger. However, this potential is still limited by the lack of rich function families. For this work, we started by identifying the limitations of the current state-of-the-art approaches which, in its turn, allowed us to consider a new threat model for FE schemes. To the best of our knowledge, we here present the first attempt to quantify the leakage during the execution of an FE scheme. By leveraging the functionality offered by Trusted Execution Environments, we propose a construction that given any message-private functional encryption scheme yields a function-private one. Finally, we argue in favour of our construction's applicability on constrained devices by showing that it has low storage and computation costs.

Keywords: Cloud Security · Forward Privacy · Functional Encryption · Function-Hiding

1 Introduction

One of the significant leaps in cryptography in the last decade was the switch from traditional encryption schemes to schemes treating encrypted data as plaintext. Although its practical application is still lagging behind, in the ensuing years this technology is expected to scale up to having a global impact.

Functional Encryption (FE) is a cryptographic scheme that allows selective computations over encrypted data. FE schemes use a key generation algorithm that outputs decryption keys with remarkable capabilities. More precisely, each decryption key sk_f is associated with a function f. In contrast to traditional cryptographic techniques, using sk_f on a ciphertext $\mathsf{Enc}(x)$ does *not* recover

x but a function $f(x)$ – thus keeping the actual value x private. While the first constructions of FE allowed the computation of a function over a *single* ciphertext, more recent works [1, 23] introduced a more general notion of multi-input FE (MIFE). In a MIFE scheme, given ciphertexts $\mathsf{Enc}(x_1), \ldots, \mathsf{Enc}(x_n)$, a user can use sk_f to recover $f(x_1, \ldots, x_n)$. The function f can allow only highly processed forms of data to be learned by the functional key holder. Unfortunately, while MIFE seems to be a perfect fit for many real-life applications – especially cloud-based ones where multiple users store large volumes of data in remote and possibly corrupted entities – most of the works in the field revolve around constructing *generic* schemes that do not support specific functions. We however believe, that despite the scientific challenges, in the near future, we will start seeing more nuance in the functions supported by FE schemes.

Having identified the importance of FE and believing that only a family of modern encryption schemes could thread the way through this, as yet, uncharted technological territory, we started looking into new and unexplored problems that may arise from their use. As the research on FE evolved, we realized that developing FE schemes themselves is only a *part of the challenge* – possibly, not even the hardest part. At first we focused on the important, till now overlooked, problem of *function-privacy* (or function-hiding). That is, how to make sure that the cloud service provider (CSP) who is performing a functional decryption, will output the correct result, without learning anything about the computed function. Moving forward, another important challenge, entirely absent from the literature, came into view. Namely, how to classify ciphertexts with regards to the functions that can be applied to them. In the standard FE model, the function is applied to the entirety of the users' data. However, in many cases this may be extremely problematic, as the function may not be defined over some of the data.

Function-Privacy: FE schemes can be either symmetric [7] or asymmetric [9]. The problem of function-privacy needs to be treated differently for each setting.

Public-key vs Private-key: Function-privacy in the public-key setting and in the private-key setting are two notions that unfortunately differ dramatically. The difference derives from the fact that in a public-key FE scheme, given a functional key sk_f for a function f, a malicious user can use the public-key to encrypt any message x of her choice and then evaluate $f(x)$. In other words, it is possible to evaluate the function f on infinitely many points – a process that reveals non-trivial information about f. However, in the private-key setting where the private key is hidden, this attack cannot work as the malicious user lacks the ability to encrypt messages. Hence, it is possible to formulate a level of security in which nothing else beyond $f(x)$ is leaked.

Function Privacy in Private-key FE: Function Privacy in the private-key setting was formalized in [17], where authors proposed a method to hide the description of the function from the evaluator. Since then, and due to the lack of a rich function family, researchers are trying to deploy function-privacy techniques for the inner-product functionality [3, 10, 11, 24, 29] . These works are focusing on functions of the form $f(x, y) = \langle x, y \rangle$, where x is provided by the user and y takes

the role of the functionality. Hence, given an encryption $\mathsf{Enc}(x)$ of some x and a functional key sk_y for y, the decryption algorithm outputs $\langle x, y \rangle$. The notion of function-privacy in these approaches comes from generating sk_y in such a way that information about y remains hidden. However sophisticated, the problem with these approaches is that the evaluator always knows that an inner-product functionality is being computed – even when both x and y remain private.

1.1 Motivation

The motivation for this work mainly derives from the lack of realistic solutions to the problem of function-privacy as well as from the identified limitations of the existing approaches. Since its formalization in [17], the main idea behind function-privacy is to encrypt the description of a function f using a semantically secure private key encryption scheme SKE, before issuing a query for f to the CSP. When the CSP receives a query, it runs the FE decryption algorithm for a universal function U which first decrypts the description of a function f and immediately applies f on an input x. While the main idea of the used subterfuge is valid, a formal definition for the description of the function is totally absent from both [17] and subsequent works [16,25,26]. This lack of formalization raises multiple concerns and limitations: *(1)* From a theoretical point of view, not formalizing the description of the function f, implies that the user's query to the CSP may also be not well-defined (as the description is part of the query). This is very problematic since in an interactive protocol, the messages exchanged between the parties tend to leak sensitive information. Hence, not providing a strict definition of the function description, makes quantifying the leakage during the scheme's execution impossible. *(2)* From a practical point of view, the problem of the actual execution of the decryption algorithm is not discussed. What does it mean from the CSP's point of view to run the function f based on the description of f and how will it run an executable given on a theoretical function description?

During the observation of the aforementioned limitations, and while we were trying to find a way to bypass them, to our surprise we also stumbled upon a new problem that is relevant to all FE schemes[4]. The observation that the FE scheme is defined so that the decryption algorithm runs on the entirety of the user's ciphertexts, for a function f, is naturally followed by a relevant question: *"What if the user, updates her collection of data by adding a ciphertext that does not live in the domain where function f is defined?"*. Such a scenario could lead the entire system to an error.

Contributions While designing new FE schemes that support a wide range of functions is of paramount importance, we also need to carefully maintain the balance between offering functionality and satisfying the properties of security. To this end, we borrow definitions and terminology from special sub-classes of

[4] Untargeted driven research is sometimes the best way to glean genuinely new insights.

FE, such as symmetric searchable encryption (SSE) [8, 20]. This is the main focus of this paper – a work that we believe will eventually allow us to successfully venture into the new. The contribution of this work can be summarized as follows: *(1)* We define a novel adversarial model applicable to any FE scheme. To do so, we consider certain information that may leak during the run of an FE scheme. To the best of our knowledge, this is the first attempt to quantify the leakage in FE. *(2)* We show how to turn *any* message-private multi-input FE scheme to a function-private one. *(3)* Through a theoretical evaluation, we show that our construction can be successfully used by constrained devices due to its low storage and computational costs. *(4)* We conducted extensive experiments to test the practicality of our construction.

2 Related Work

The first notions of Functional Encryption appeared as part of the work presented in [12], where authors constructed the first Identity-Based Encryption (IBE) scheme. Later, it was formally defined as a generalization of public-key encryption in [15]. Since then, numerous studies with general definitions and generic constructions of FE have been proposed [16, 23, 27, 30], both in the public, and private-key settings. However, despite the promising works that have been published, there is a clear lack of work proposing FE schemes supporting specific functions – a necessary step towards allowing the FE to transcend its limitations and provide the foundations for reaching its full potential. To the best of our knowledge, currently the number of supported functionalities is limited to inner products [2–4] and quadratic polynomials [28].

Despite the absence of rich function families in FE, researchers have already set the ground for stronger security guarantees that would prevent the decryptor from learning the computed function. The problem of function-privacy in the public-key setting, where extra restrictions need to be posed to the adversary thus resulting in weaker threat models, was first studied in [13, 14] and [5]. Currently, the trend of function-privacy in the public key setting, revolves around obfuscation [6, 18, 22, 23, 25], which seems to be the only possible way to bypass the limitations of public-key schemes discussed in Section 1. In the private-key setting, a first general framework was presented in [5], followed by a strict formalization in [17]. In theory, in the private-key setting, it is possible to achieve a function-privacy scheme by relying solely on the semantic security of a symmetric key encryption scheme. However, the lack of rich function families, prevents researchers from digging deeper in the problem of function-privacy. Due to this, the majority of the literature relies on function-privacy techniques to partially hide information used in the computation of inner-product functionalities [3, 10, 11, 24, 29].

3 Background

In this section, we introduce our notation and recall the definitions of MIFE in the private-key setting.

Notation A function $negl(\cdot)$ is called negligible, iff $\forall c \in \mathbb{N}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, negl(n) < n^{-c}$. A probabilistic polynomial time (PPT) adversary $\mathcal{ADV}$ is a randomized algorithm for which there exists a polynomial $p(z)$ such that for all input z, the running time of $\mathcal{ADV}(z)$ is bounded by $p(|z|)$.

Definition 1. *Let $\mathcal{F}$ be a rich family of functions such that every $f \in \mathcal{F}$ is defined as $f : \mathcal{X}_1 \times \ldots \times \mathcal{X}_n \to \mathcal{Y}$. A MIFE scheme for $\mathcal{F}$ consists of the following four algorithms:*

- $\mathsf{Setup}(1^\lambda)$ *takes as input the unary representation of the security parameter λ and outputs a master secret key* msk.
- $\mathsf{Enc}(\mathsf{msk}, x_i)$ *takes as input the master secret key* msk *and a message x_i and outputs a ciphertext c_i.*
- $\mathsf{KeyGen}(\mathsf{msk}, f)$ *takes as input the master secret key* msk *and a function f, and outputs a decryption key sk_f for f.*
- $\mathsf{Dec}(\mathsf{sk}_f, c_1, \ldots, c_n)$ *takes as input a decryption key sk_f for a function f and n ciphertexts, and outputs a value $y \in \mathcal{Y}$.*

Correctness *For correctness we require that for every $f \in \mathcal{F}$ and for every $x_i \in \mathcal{X}_i$ it holds:*

$$\mathsf{Dec}(\mathsf{KeyGen}(\mathsf{msk}, \mathsf{f}), \mathsf{Enc}(\mathsf{msk}, x_i), \ldots, \mathsf{Enc}(\mathsf{msk}, x_n)) = y \in \mathcal{Y},$$

with all but a negligible probability over the internal randomness of $\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}$ *and* Dec.

Two desirable properties of FE schemes are those of message-privacy and function-privacy.

Definition 2 (Message-Privacy). *An FE scheme is said to be message-private, if any two messages x_0 and x_1 are computationally indistinguishable for any adversary that can adaptively obtain decryption keys for any function such that $f(x_0) = f(x_1)$.*

Naturally, the above definition is quite restrictive when we deal with a rich family of functions. To this end, in [17], authors proposed a stronger security definition for function-privacy.

Definition 3 (Function-Privacy). *An FE scheme is said to be function-private if any adversary that obtains encryptions of x_0 and x_1, and functional decryption keys $\mathsf{sk}_{f_0}, \mathsf{sk}_{f_1}$ for two functions f_0, f_1 such that $f_0(x_0) = f_1(x_1)$, cannot distinguish between them.*

For a formal presentation of the security games for message and function privacy, we refer the reader to [17].

4 Architecture

In this section, we present the topology of our construction. In particular, we assume the existence of the three following entities:
Data Owner: The data owner (u_i) keeps a list of unique identifiers for all possible functions supported by the underlying MIFE scheme. In addition to that, u_i is responsible for encrypting her data and creating the necessary indexes. In particular, u_i first parses her data locally before outsourcing it to the CSP. During this process, she generates the following three indexes:

- $\mathcal{D}$ – a dictionary containing mappings between function identifiers and ciphertexts that can be given as input to each function.
- $\mathsf{No.Runs}[f]$ which contains mappings between the unique identifier of each function f, and the number of times f has already been executed.
- $\mathsf{No.Inputs}[f]$ which consists of mappings between function identifiers and the number of inputs for each function f.

$\mathsf{No.Runs}[f]$ and $\mathsf{No.Inputs}[f]$ are kept locally on u_i's side, while $\mathcal{D}$ is outsourced to the CSP.
Cloud Service Provider (CSP) The CSP storage will consist of the ciphertexts as well as the dictionary $\mathcal{D}$. Each $\mathcal{D}$ entry is encrypted under a different temporary key $\mathsf{tk_f}$. Thus, given $\mathsf{tk_f}$ and the number of inputs for a function f, the CSP can recover all the ciphertexts that will be given as input to the MIFE decryption algorithm.
Secure Component (SC): SC is a TEE-protected component that resides in the CSP. We assume that during the deployment phase, SC receives a series of function descriptions, along with their executables. SC is triggered by the CSP, upon receiving a query for a function by a user u_i. SC is responsible for running the MIFE decryption algorithm upon receiving a collection of ciphertexts from the CSP, a functional decryption key sk_f for a function f and the unique identifier of f. SC has the following properties:

Attestation We assume that the TEE offers the feature of attestation. In particular, during attestation, SC generates an attestation report that can be verified by any party. The report is signed with a private key and hence, anyone holding the public key will be in position to verify it.

Sealing When data are stored in untrusted memory, they are encrypted with a key known only to SC. Sealed data can be recovered even if SC is required to restart.

5 Formal Construction

This section constitutes the core contribution of our paper, as we formally define our construction. For the needs of our construction, we assume the existence of the following building blocks:

1. A key-private MIFE scheme that fulfils the property of message privacy.
2. A first and second pre-image resistant hash function H.
3. A pseudorandom function $G : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$
4. An IND-CCA2 secure public-key cryptosystem PKE such that: $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.
5. An EUF-CMA secure signature scheme $\mathsf{S} = (\mathsf{sign}, \mathsf{ver})$.
6. A synchronized clock between all the parties.

Note here, that we only require the MIFE scheme to be message-private. Hence, our construction can be applied to any message-private MIFE scheme, to further enhance it with the property of function-privacy.

5.1 Secure Hardware Formalization

During the execution of the protocol, we assume the existence of a secure hardware (SHW) component SC. In the beginning, $\mathsf{SHW.Setup}$ runs to generate a public/secret key pair. SC is then initialized, by loading a program Q and producing an identifier SC. This is done by running $\mathsf{SHW.Load}$. After the initialization of SC, $\mathsf{HW.Run}$ is executed with different inputs. Finally, SC runs $\mathsf{SHW.Run\&Report}$ to generate a signed attestation repost.

- $\mathsf{SHW.Setup}(1^\lambda)$: Takes an input a security parameter λ and produces a public/secret key pair $(\mathsf{pk_{rpt}}, \mathsf{sk_{rpt}})$, used to sign and verify reports.
- $\mathsf{SHW.Load}(Q)$: Takes as input a program Q. A private region of memory, dedicated to execute Q, is allocated. Moreover, an identifier id is created that will be used to identify this particular region of memory.
- $\mathsf{SHW.Run}(\mathsf{SC}, in)$: Takes as input an identifier id and some input in. It then runs the program in the memory region specified by id, with in as input.
- $\mathsf{SHW.Run\&Report}(\mathsf{id}, in)$: Takes as input an identifier id and some input in and outputs an attestation report rpt signed with $\mathsf{sk_r pt}$. The report is verifiable by any party (local or remote) holding the corresponding public key $\mathsf{pk_r pt}$.

5.2 Forward and Function Private MIFE

We are now ready to proceed with the formal definition of FFP (Forward and Function Private). Our construction consists of four polynomial time algorithms such that $\mathsf{FFP} = (\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$.

FFP.KeyGen*:* During the execution of KeyGen, the data owner invokes $\mathsf{MIFE.Setup}$ to generate a master secret key msk for the message-private MIFE scheme. Apart from that, a key K_G for a PRF G is generated.

FFP.Setup*:* This is a two-party protocol between a data owner u_i and the CSP. The protocol is initiated by u_i who wishes to encrypt her data. First, u_i locally creates mappings between a sequence of data $(x_1, \ldots, x_n)$ and function descriptions. These mappings, simply specify the proper inputs for each function f_i. As a next step, u_i computes a temporary key for each function as $\mathsf{tk_f} = G(\mathsf{K_G}, f||\mathsf{No.Runs}[f])$. This key will allow u_i to start building the dictionary $\mathcal{D}$. In particular, for each (x_i, f_i) pair, u_i first invokes MIFE.Enc to encrypt x_i, resulting in a ciphertext c_i, and then further masks the function description as $mf_i = H(\mathsf{tk_f}||\mathsf{No.Inputs}[f])$. Then, the pair $\{c_i, mf_i\}$ is added to the dictionary $\mathcal{D}$. The data owner u_i repeats these steps for all $x_i \in (x_1, \ldots, x_n)$. Finally, $\mathcal{D}$ is outsourced to the CSP via $m_1 = \langle t_1, \mathcal{D}, \sigma_{u_i}(h(t_1||\mathcal{D}))\rangle$ to the CSP, where t_1 is a timestamp and σ_{u_i} denotes the signature of u_i. Upon reception, the CSP can verify u_i's signature, the freshness and the integrity of the m_1. FFP.Setup is presented in detail in Algorithm 1.

Algorithm 1 FFP.Setup

Data Owner:
1: **for all** (x_i, f_i) pairs **do**
2: **if** $\mathsf{No.Inputs}[f_i] = \text{null}$ **then**
3: $\mathsf{No.Inputs}[f_i] = 0$
4: **if** $\mathsf{No.Runs}[f_i] = \text{null}$ **then**
5: $\mathsf{No.Runs}[f_i] = 0$
6: $\mathsf{No.Inputs}[f_i] + +$
7: $\mathsf{tk}_{f_i} = G(\mathsf{K_G}, \mathsf{f_i}||\mathsf{No.Inputs}[\mathsf{f_i}])$ ▷ temporary key for f_i
8: $mf_i = h(\mathsf{tk_{f_i}}||\mathsf{No.Inputs}[f)i])$ ▷ mask the function
9: $c_i = \mathsf{MIFE.Enc}(\mathsf{msk}, \mathsf{x_i})$
10: Add the (c_i, mf_i) pair into $\mathcal{D}$
11: Outsource $\mathcal{D}$ to the CSP via $m_1 = \langle t_1, \mathcal{D}, \sigma_{u_i}(h(t_1||\mathcal{D}))\rangle$
12: Store $\mathsf{No.Inputs}[f]$ and $\mathsf{No.Runs}[f]$ locally

FFP.Add*:* This is a two-party protocol between u_i and the CSP. The procedure is identical to that of FFP.Setup but is applied to only one (x_i, f_i) pair. To add the pair to the dictionary $\mathcal{D}$, u_i generates an add token as $\tau_a(x_i, f_i) = (c_i, mf_i)$, where c_i and mf_i are generated as discussed in FFP.Setup. Finally, u_i sends $m_2 = \langle t_2, (c_i, mf_i), \sigma_{u_i}(h(t_2||(c_i, mf_i)))\rangle$ to the CSP. We omit a formal description as it is identical to that of Algorithm 1, without the for loop in line 1.

FFP.Query*:* This is a three-party protocol between u_i, the CSP and SC. Assume now that u_i wishes to run a function $f \in \mathcal{F}$, on the encrypted data $(x_1, \ldots, x_n)$. To do so, u_i first needs to generate a query token for the function f by following a series of steps:

Step 1. Invokes MIFE.KeyGen to generate a decryption key sk_f.

Step 2. Retrieves the $\mathsf{No.Runs}[f]$ and $\mathsf{No.Inputs}[f]$ values from her local indexes and calculates the masked version of the function description as described in $\mathsf{FFP.Setup}$.

Step 3. Now, u_i is required to calculate the updated value of mf. To do so, increments the $\mathsf{No.Runs}[f]$ values by one, and computes a new temporary key $\mathsf{tk_f}$. Based on this temporary key, u_i calculates the new masked version of f, mf'.

After completing these steps, u_i can simply calculate the query token $\tau_q(f) = (\mathsf{sk}_f||mf_i||\mathsf{No.Inputs}[f]||mf'||h(f||\mathsf{No.Runs}[f])))$ which is also forwarded to the CSP via $m_3 = \langle t_3, \mathsf{Enc}_{pk_{SC}}(\mathsf{sk}_f), h(f||\mathsf{No.Inputs}[f]), \tau_q(f), \sigma_{u_i}(h')\rangle$ where h' is the following hash: $h(t_3||\mathsf{Enc}_{pk_{SC}}(\mathsf{sk}_f)||h(f||\mathsf{No.Inputs}[f])||\tau_q(f))$. Upon reception, the CSP uses $\tau_q(f)$ to locate all ciphertexts $(c_1, \ldots, c_n)$ that can be given as input to f_i and sends them to SC along with sk_f and $h(f||\mathsf{No.Runs}[f])$ via $m_4 = \langle t_4, Res, m_3, \sigma_{CSP}(h(t_4, Res, m_3))\rangle$. Apart from that, the CSP removes all the current masked versions of f, and replaces them with the ones in $\mathcal{L}$. SC, knowing the hash of the unique id of the executable corresponding to the function f, can locate which is the proper executable. As a result, SC invokes $\mathsf{MIFE.Dec}$ and outputs $\mathsf{out} = f(x_1, \ldots, x_n)$ which is sent back to u_i to the CSP via $m_5 = \langle t_5, \mathsf{out}, \sigma_{SC}(h(t_5||\mathsf{out}))\rangle$. Finally, the CSP forwards m_5 to u_i. $\mathsf{FFP.Query}$ is formally presented in Algorithm 2.

Algorithm 2 Query

Data Owner:
$\mathsf{sk}_f \leftarrow \mathsf{MIFE.KeyGen}(\mathsf{skf}, f)$ ▷ generate the decryption key
$\mathsf{tk_f} = G(\mathsf{K}_G, f||\mathsf{No.Runs}[f])$
$mf' = h(\mathsf{tk_{f_i}}||\mathsf{No.Inputs}[f])$
$\mathsf{No.Runs}[f] + +$
$\mathsf{tk'_f} = G(\mathsf{K}_G, f||\mathsf{No.Runs}[f])$
$mf'_i = h(\mathsf{tk'_f}||\mathsf{No.Inputs}[f])$
$\tau_q(f) = (\mathsf{sk}_f, mf, \mathsf{No.Inputs}[f], mf', h(f||\mathsf{No.Runs}[f]))$
Send m_3 to the CSP
CSP:
$Res = \{\}$
for $i = 1$ **to** $i = \ell$ **do** ▷ $\mathsf{No.Inputs}[f] = \ell$
    $c_i = \mathcal{D}(mf, :)$ ▷ find the input ciphertext based on the masked description
    $Res = Res \cup c_i$
    Replace mf with mf'
Send $m_4 = \langle t_4, Res, m_3, \sigma_{CSP}(h(t_4, Res, m_3))\rangle$ to SC
SC:
Find which executable corresponds to the function description
Update the stored function description by incrementing $\mathsf{No.runs}$ by one
Run $\mathsf{MIFE.Dec}(\mathsf{sk}_f, c_1, \ldots, c_\ell)$
Output $\mathsf{out} = f(x_1, \ldots, x_\ell)$

19: Send $m_5 = \langle t_5, \mathsf{out}, \sigma_{SC}(h(t_5||\mathsf{out}))\rangle$ to the CSP
CSP:
20: Forward m_5 to the Data Owner

5.3 Leakage Functions

Based on our formal construction, we can now formally quantify the total leakage by providing a concrete definition of the leakage function $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{query}, \mathcal{L}_{add})$. This is an essential part of our work as this function will be explicitly used during the proof of security in the next section.

- $\mathcal{L}_{keygen} = \lambda$. The leakage function associated with FFP.KeyGen is restricted to the length of the generated keys.
- $\mathcal{L}_{setup} = (N, m)$, where N is the total number of mappings in $\mathcal{D}$, and m is the total number of ciphertexts. The leakage function associated with the FFP.Setup leaks the size of $\mathcal{D}$, and the number of ciphertexts.
- $\mathcal{L}_{query} = (qp[t], ap, |\mathsf{sk}_f|, |h_{out}|, \mathsf{out})$. The leakage function associated with FFP.Query leaks the query and access patterns, the size of the functional key, the outputs of the hash function h and the output $\mathsf{out} = f(x_1, \ldots, x_n)$.
- $\mathcal{L}_{add} = (|c_i|, |f_i|)$. The leakage function associated with FFP.Add leaks the sizes of the added ciphertext and the function description.

Moreover, note that even if we use a different temporary key $\mathsf{tk_f}$ for each query, the query pattern is still leaked since the CSP can easily guess that newly re-keyed entries correspond to the input of the queried function. The temporary keys are not used to hide the query pattern, but to provide forward privacy as per Definition 5.

5.4 Secure Component Analysis

While in the previous section, we provided a detailed presentation of FFP, our approach was agnostic to the TEE-enabled component. In this section, we aim to formalize the behaviour of SC by providing an analysis of the programs that are executed by SC. This will help us better understand the role of SC and, subsequently, formalize its security properties. SC is initialized during FFP.Setup as follows: **Initialization**: SC is initialized by generating a public/secret and singing/verification key pairs. To do so, a program $\mathbf{Q^{init}}$ is loaded:

$\mathbf{Q^{init}}$:

- On input ("*initialize*", 1^λ):
  1. Run $(\mathsf{pk_{SC}}, \mathsf{sk_{SC}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$.
  2. Publish pk.

  Run $\mathsf{SC} \leftarrow \mathsf{SHW.Load}(\mathsf{Q^{init}})$.

Run: SC is required to run during the execution of FFP.Query to produce an output out. Moreover, SC signs out since it also needs to generate an attestation report. This is done by a program $\mathbf{Q^{run}}$ as follows:

$\mathbf{Q^{run}}$:

- On input $(\text{“}run\text{”}, m_4)$:
  1. Open m_4; Verify the freshness, the integrity of the message and the signature of the CSP; If the verification fails, output $\perp$.
  2. Compute $\mathsf{sk}_f = \mathsf{Dec}(\mathsf{sk}_{SC}, \mathsf{Enc}_{sk_{SC}}(\mathsf{sk}_f))$.
  3. Compute and output m_5

  Run $m_5 \leftarrow \mathsf{SHW.Run}(\mathsf{SC}, (\text{“}\mathsf{run}\text{”}, \mathsf{m_4}))$ and then:
  $\mathsf{rpt} \leftarrow \mathsf{SHW.Run\&Report}(SC, (\text{“}run\text{”}, m_4))$

6 A Novel Adversarial Model

To capture our new notion of security in FE we rely on the real experiment against ideal experiment formalization. In particular, in the real experiment the adversary $\mathcal{ADV}$ observes the algorithms being executed honestly, while in the ideal experiment a simulator $\mathcal{S}$ simulates the functionalities based on explicit leakage. The leakage is formalized by a function $\mathcal{L}$ such that $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{query}, \mathcal{L}_{add})$ where each component corresponds to the leakage associated with the key generation, setup, query and addition algorithms. The idea is the following: $\mathcal{ADV}$ has full control of the client. Thus, she can trigger any operation. $\mathcal{ADV}$ issues a polynomial number of queries, and for each query she records the output. The scheme is said to be $\mathcal{L}$-adaptively secure if $\mathcal{ADV}$ cannot distinguish between the real and the ideal experiments.

Definition 4. *($\mathcal{L}$-Adaptive Security) Let FFP = (KeyGen, Setup, Query, Add) be as defined in Section 5. Moreover, let $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{query}, \mathcal{L}_{add})$ be the leakage function of FFP. We consider the following experiments between an adversary $\mathcal{ADV}$ and a challenger $\mathcal{C}$:*

$\mathbf{Real}_{\mathcal{ADV}}(\lambda)$

$\mathcal{ADV}$ outputs a set of inputs for various functions from a rich function family $\mathcal{F}$. $\mathcal{C}$ generates a key msk*, and runs* FFP.Setup*. $\mathcal{ADV}$ then makes a polynomial number of adaptive queries $q = \{f, (x_1, f_1)\}$. For each q, she receives back either a query token for f, $\tau_q(f)$, or an add token $\tau_a(x_1, f_1)$ for the pair (x_1, f_1) Finally, $\mathcal{ADV}$ outputs a bit b.*

$\mathbf{Ideal}_{\mathcal{ADV},\mathcal{S}}(\lambda)$

$\mathcal{ADV}$ outputs a set of inputs for various functions from a rich function family $\mathcal{F}$. $\mathcal{S}$ gets $\mathcal{L}_{\mathsf{setup}}$ to simulate Setup*. $q = \{f, (x_1, f_1)\}$. For each q, $\mathcal{S}$ is given $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{add}, \mathcal{L}_{query})$. $\mathcal{S}$ then simulates the dictionary, the tokens and, in the case of addition, a ciphertext. Finally, $\mathcal{ADV}$ outputs a bit b.*

We say that FFP is secure if $\forall$ PPT $\mathcal{ADV}$, $\exists$ $\mathcal{S}$ such that:

$$|Pr[(\mathsf{Real}_{\mathcal{ADV}}) = 1] - Pr[(\mathsf{Ideal}_{\mathcal{ADV},\mathcal{S}}) = 1]| \leq negl(\lambda) \quad (1)$$

Moreover, we further require that from the CSP's point of view, newly added (x_i, f_i) pairs do not leak any information about past queries. This property, makes adaptive attacks a lot less effective [32]. This requirement, forces us to define the property of *Forward Privacy* for FE schemes. Informally, we will say that a scheme is said to be *forward private* if for all file insertions that take place after the successful execution of the Setup algorithm, the leakage is limited to the sizes of the function's input and the function's description. More formally:

Definition 5 (Forward Privacy). *An $\mathcal{L}$-adaptively FFP scheme is forward private iff the leakage function $\mathcal{L}_{Add}$ can be written as:*

$$\mathcal{L}_{\mathsf{add}}(x_i, f_i) = (|x_i|, |f_i|) \tag{2}$$

For the following two definitions, let us assume that the data owner has already issued t queries $\mathcal{Q} = \{q_1, \ldots, q_t\}$, up to time t.

Definition 6 (Query Pattern). *The query pattern is defined to be a vector QP that shows which query each function corresponds to. For example, $\mathsf{SP}[t] = f_\kappa$ means that a query for the function f_κ was issued at time t.*

Definition 7 (Access Pattern). *We define the Access Pattern to be the set of all ciphertexts that can be given as input to a function f at time t. The set is denoted by $\mathsf{AP}_{f,t}$.*

7 Security Analysis

We are now ready to prove the security of our construction against the threat model defined in Section 6. To do so, we need to prove the existence of a simulator $\mathcal{S}$, that given the leakage function $\mathcal{L}$, simulates a perfect view for an adversary $\mathcal{ADV}$ with all but a negligible probability. Before we proceed with the formal proof of security, we present a high-level overview:

Proof Sketch: We will rely on a constructive proof in the sense that we will prove the existence of $\mathcal{S}$, by constructing it. This will be achieved through a hybrid argument. More precisely, we will design five hybrids $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$ and $\mathcal{H}_4$ such that $\mathcal{H}_0$ is the real experiment and $\mathcal{H}_4$ the ideal one, as they were defined in Section 6. Each hybrid $\mathcal{H}_i$, will be constructed by replacing a real functionality with a simulated one given the corresponding leakage function $\mathcal{L}_i$. Our goal is to prove that no PPT adversary $\mathcal{ADV}$ will be able to distinguish between two consecutive hybrids $\mathcal{H}_i, \mathcal{H}_{i+1}$. The hybrids are illustrated in Table 1.

Table 1: Hybrid Description

Hybrid	Description
$\mathcal{H}_0$	This is the real experiment
$\mathcal{H}_1$	Simulate FFP.KeyGen given $\mathcal{L}_{keygen}$
$\mathcal{H}_2$	Simulate FFP.Setup given $\mathcal{L}_{setup}$
$\mathcal{H}_3$	Simulate FFP.Add given $\mathcal{L}_{add}$
$\mathcal{H}_4$	This is the ideal experiment

It is important to note that simulation-based security definitions for FE already exist [15,23]. However, in contrast with these works, we address a different problem, as we are more interested in proving the security of a general framework and not that of a FE scheme.

Theorem 1. *Let MIFE be a message-private multi-input functional encryption scheme. Moreover, let G be a secure pseudorandom function and h a first and second pre-image resistant hash function. Then* $\mathsf{FFP} = (\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$*, with corresponding leakage function* $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{query}, \mathcal{L}_{add})$ *is $\mathcal{L}$-adaptively secure as per Definition 4.*

Proof. We construct a simulator $\mathcal{S}$ that simulates the real functionality in such a way that no PPT adversary $\mathcal{ADV}$ can distinguish between the real and ideal worlds. Note here that $\mathcal{ADV}$ only expects an output from FFP.Query and in this case, apart from simulating the search token, $\mathcal{S}$ also needs to simulate the process executed by SC. $\mathcal{S}$ is given as input $\mathcal{L} = (\mathcal{L}_{keygen}, \mathcal{L}_{setup}, \mathcal{L}_{query}, \mathcal{L}_{add})$.

In a pre-processing phase, $\mathcal{S}$ runs $\mathsf{SHW.Setup}(1^\lambda)$ and records $(\mathsf{pk}_{\mathsf{rpt}}, \mathsf{sk}_{\mathsf{rpt}})$ generated during the process.

Hybrid 0 $(\mathcal{H}_0)$ This is the real world.

Hybrid 1 $(\mathcal{H}_1)$ Like $\mathcal{H}_0$, but instead or FFP.KeyGen, $\mathcal{S}$ gets as input $\mathcal{L}_{keygen}$ and simulates the real world.

Claim. $\mathcal{H}_0$ is computationally indistinguishable from $\mathcal{H}_1$.

The proof of Claim 7 is straightforward as $\mathcal{ADV}$ does not have access to any key generated during this phase. Hence, we can safely assume that:

$$|Pr[(\mathcal{H}_0) = 1] - Pr[(\mathcal{H}_1) = 1]| \leq negl(\lambda) \tag{3}$$

Hybrid 2 $(\mathcal{H}_2)$ Like $\mathcal{H}_1$, but instead or FFP.Setup, $\mathcal{S}$ gets as input $\mathcal{L}_{setup}$ and simulates the real world.

Claim. $\mathcal{H}_1$ is computationally indistinguishable from $\mathcal{H}_2$.

Proof. $\mathcal{S}$ is given $\mathcal{L}_{setup} = (N, m)$ and proceeds as shown in Algorithm 3.

Since $\mathcal{L}_{setup} = (N, m)$, $\mathcal{S}$ knows exactly both the size of the dictionary $\mathcal{D}$, as well as the number of resulted ciphertexts. Moreover, as we can assume the hash

Algorithm 3 Setup Simulation

procedure $\mathcal{S}(\mathcal{L}_{setup})$
  $k \leftarrow \mathsf{MIFE.KeyGen}(1^\lambda)$
  for $i = 1$ **to** $i = N$ **do**
    Output a simulated masked function description sim_{f_i}
    Simulate the ciphertexts as $sim_{c_i} \leftarrow \mathsf{MIFE.Enc}(k, x_j)$
    Store all (sim_{f_i}, sim_{c_i}) in a dictionary $\mathcal{D}_\mathcal{S}$
  Create a dictionary Keys to store the last temporary key for each function query.
  Create a dictionary $\mathcal{O}$ to reply to random oracle queries.
  Create a dictionary DecKeys to store the functional decryption keys for different functions.
  Output $\mathcal{D}_\mathcal{S}$

function h and the security parameter λ to be publicly known, then $\mathcal{S}$ knows the exact sizes of the real masked functions mf_i and ciphertexts c_i. Hence it is straightforward to generate simulations of both the masked functions sim_{f_i} and the ciphertexts sim_{c_i} of correct sizes. What remains to be done, is show that $\mathcal{ADV}$ cannot distinguish between an encryption of a plainetext x_i and that of a random x_j such that $f(x_i) = f(x_j)$ and $|x_i| = |x_j|$. However, recall that we have assumed that MIFE is message-private, which means that an adversary cannot distinguish between the encryptions of two messages x_i and x_j even is she possess functional decryption keys. Hence, we conclude that the encryption of a x_i and that of a x_j such that $f(x_i) = f(x_j)$ are computationally indistinguishable and hence:

$$|Pr[(\mathcal{H}_0) = 1] - Pr[(\mathcal{H}_1) = 1]| \leq negl(\lambda) \tag{4}$$

Hybrid 3 $(\mathcal{H}_3)$ Like $\mathcal{H}_2$, but instead of FFP.Add, $\mathcal{S}$ gets as input $\mathcal{L}_{add}$ and simulates the real world.

Claim. $\mathcal{H}_2$ is computationally indistinguishable from $\mathcal{H}_3$.

Proof. $\mathcal{S}$ is given as input $\mathcal{L}_{add}$ and proceeds as shown in Algorithm 4:

Algorithm 4 Addition Simulation

procedure $\mathcal{S}(\mathcal{L}_{add})$
  $L = \{\}$
  $sim_{c_i} \leftarrow \mathsf{MIFE.Enc}(\mathsf{k}, x_j)$ for a random x_j such that $f(x_i) = f(x_j)$
  Simulate a (sim_{f_i}, sim_{c_i}) pair
  Output $\tau_\alpha = (c_i, mf_i)$

Since $\mathcal{L}_{add} = |c_i|, |f_i|$, $\mathcal{S}$ can easily generate a simulated ciphertext sim_{c_i} of the correct size. Similarly, $\mathcal{S}$ generates correctly a simulated masked function description of the correct size. Finally, once again, the message-privacy property of

the MIFE scheme, ensures us that $\mathcal{ADV}$ cannot distinguish between the encryptions of an actual plaintext x_i, and that of a random x_j such that $f(x_i) = f(x_j)$. As a result:

$$|Pr[(\mathcal{H}_2) = 1] - Pr[(\mathcal{H}_3) = 1]| \leq negl(\lambda) \tag{5}$$

Moreover, note that since we successfully simulated FFP.Add given only the leakage function $\mathcal{L}_{add}$, we also proved that FFP preserves our newly defined property of forward privacy in functional encryption.

Hybrid 4 $(\mathcal{H}_4)$ Like $\mathcal{H}_3$, but instead or FFP.Query, $\mathcal{S}$ gets as input $\mathcal{L}_{query}$ and simulates the real world.

Claim. $\mathcal{H}_2$ is computationally indistinguishable from $\mathcal{H}_3$.

Proof. Proving Claim 7 is trickier that the previous ones as addition operations affect query operations. Hence, $\mathcal{S}$ needs to be constructed in such a way that it keeps track of all the additions in order to respond consistently to $\mathcal{ADV}$. To achieve this, we need to design three more dictionaries:

- Keys$[f]$, that keeps track of the last temporary key that was used for the function f.
- $\mathcal{O}[\mathsf{tk_f}][i], i \in \{0, 1\}$, which serves as a random oracle responsible to provide $\mathcal{ADV}$ with consistent query tokens. More precisely, $\mathcal{O}[\mathsf{tk_f}][0]$ stores the simulated masked function, while $\mathcal{O}[\mathsf{tk_f}]$ stores the simulated ciphertext that is required to recover c_i.
- DecKeys, that keeps track of the functional decryption key for each function f.

$\mathcal{S}$ is given $\mathcal{L}_{query}$ and simulates the query tokens as shown in Algorithm 5.

Algorithm 5 Query Simulation

procedure $\mathcal{S}(\mathcal{L}_{query})$
  ℓ: Total number of inputs for the function f_i
  if $\mathsf{Keys}[f_i] = \text{null}$ **then**
    $\mathsf{tk_{f_i}} \leftarrow \{0,1\}^\lambda$
  $\mathsf{tk_{f_i}} = \mathsf{Keys}[f_i]$
  if $\mathsf{DecKeys}[f] = null$ **then**
    $\mathsf{sk}_f \leftarrow \{0,1\}^\lambda$
  $\mathsf{sk}_f = \mathsf{DecKeys}[f_i]$
  for $i = 1$ **to** $i = \ell$ **do**
    if $\mathcal{O}[\mathsf{tk_{f_i}}] = \text{null}$ **then**
      if x_i is added **after** Setup **then**
        Pick a $(x_i, (sim_{mf_i}, sim_{c_i}))$ pair
      else
        Pick a (sim_{mf_i}, sim_{c_i}) pair
      $\mathcal{O}[\mathsf{tk_{f_i}}][0] = sim_{mf_i}$
      $\mathcal{O}[\mathsf{tk_{f_i}}][1] = sim_{c_i} \oplus x_i$
    else
      $sim_{mf_i} = \mathcal{O}[\mathsf{tk_{f_i}}][0]$
      $sim_{c_i} = \mathcal{O}[\mathsf{tk_{f_i}}][1]$
    Delete either (sim_{mf_i}, sim_{c_i}) or $(x_i, (sim_{mf_i}, sim_{c_i}))$ from $\mathcal{D}_\mathcal{S}$.
  NewMap $= \{\}$
  $\mathsf{tk_{f_i}}' \leftarrow \{0,1\}^\lambda$
  $\mathsf{Keys}[f_i] = \mathsf{tk_{f_i}}'$
  for $i = 1$ **to** ℓ **do**
    Simulate a novel (sim_{mf_i}, sim_{c_i}) pair and add it into $\mathcal{D}$
    NewMap $=$ NewMap $\cup$ $((sim_{mf_i}, sim_{c_i}))$
    $\mathcal{O}[\mathsf{tk_f}'][0] = sim_{mf_i}$
    $\mathcal{O}[\mathsf{tk_f}'][1] = sim_{c_i}$
  output $\tau_q(f) = (\mathsf{sk}_f || sim_{mf} || \ell || \text{NewMap})$

From the description of Algorithm 5, it is clear that the simulated query token has exactly the same size and format with the real one and hence, no PPT adversary can distinguish between them. What remains to be done is for $\mathcal{S}$ to simulate SC and its response. To do so, $\mathcal{S}$ first generates a simulated m_4' message by replacing the components of the actual m_4 with the simulated ones where $m_4 =$. Having showed that $\mathcal{S}$ can successfully simulate the query token given $\mathcal{L}_{query}$, generating m_4' is straightforward. As a result, what we need to show is that:

$$\begin{aligned} &|Pr[(\mathsf{SHW.Run}(SC, (\text{“}run\text{”}, m_4))) = 1] \\ &\quad - Pr[(\mathsf{SHW.Run}(SC, (\text{“}run\text{”}, m_4'))) = 1]| \leq negl(\lambda) \end{aligned} \tag{6}$$

and then, for the attestation:

$$|Pr[(\mathsf{SHW.Run\&Report}(SC,(\text{“}run\text{”},m_4)))=1] - Pr[(\mathsf{SHW.Run\&Report}(SC,(\text{“}run\text{”},m'_4)))=1]|\leq negl(\lambda) \tag{7}$$

Proving inequalities 6 and 7, requires to prove that given m'_4, $\mathcal{S}$ can output a simulated m'_5 which is computationally indistinguishable from m_5. Recall that $m_5 = \langle t_5, \mathsf{out}, \sigma_{SC}(h(t_5||\mathsf{out}))\rangle$. However, since $\mathsf{out} \in \mathcal{L}_{query}$ and $\mathcal{S}$ already possess $(\mathsf{sk_r pt})$, it is straightforward to output a message m'_5 identical to m_5, and in this case, we have that:

$$|Pr[(\mathsf{SHW.Run}(SC,(\text{“}run\text{”},m_4)))=1] - Pr[(\mathsf{SHW.Run}(SC,(\text{“}run\text{”},m'_4)))=1]| = 0 \tag{8}$$

and

$$|Pr[(\mathsf{SHW.Run\&Report}(SC,(\text{“}run\text{”},m_4)))=1] - Pr[(\mathsf{SHW.Run\&Report}(SC,(\text{“}run\text{”},m'_4)))=1]| = 0 \tag{9}$$

Moreover, tampering with the report that is produced as part of $\mathsf{SHW.Run\&Report}$, implies that $\mathcal{ADV}$ can forge SC's signature, which can only happen with negligible probability.

Hence, we conclude that:

$$|Pr[(\mathcal{H}_3)=1] - Pr[(\mathcal{H}_4)=1]|\leq negl(\lambda) \tag{10}$$

By combining inequalities 3-10, and using the triangle inequality and the fact that the finite sum of negligible functions is still negligible, we conclude that:

$$|Pr[(\mathcal{H}_0)=1] - Pr[(\mathcal{H}_4)=1]|\leq negl(\lambda) \tag{11}$$

Note that the description of the function is kept hidden and not used at all during the entire simulation. This result is quite remarkable as it signifies that we can simulate a perfect view of the real world, by keeping the description of the function hidden. Hence, starting with any private key MIFE scheme that is also message-private, FFP can further enhance it with the properties of forward and function privacy.

Side-Channel Attacks: Recent works [19,31] have shown that TEEs are vulnerable to software attacks. However, according to [21], these attacks can be prevented if the programs running in the TEE are data-obvious. Thus, leakage can be avoided if the programs do not have memory access patterns or control flow branches that depend on the values of sensitive data.

8 Evaluation

8.1 Theoretical Evaluation

Computational Cost Despite having a larger setup time than a generic MIFE scheme, FFP is still characterized by its efficiency as it solves various complex

problems by achieving optimal query and addition costs. More precisely, a generic MIFE scheme would have a setup time of $O(n)$ where n is the total number of plaintexts to be encrypted, while FFP has a setup time of $O(N)$, where N is the total number of (x_i, f_i) pairs. Apart from that, the query cost is $O(\ell)$, where ℓ is the number of the resulted ciphertexts. This solution is superior to a naive one in which the whole dictionary would be scanned and the total complexity would be $O(N)$. Reducing the running time from $O(N)$, to $O(\ell)$ is achieved by embedding the $\mathsf{No.Inputs}[f_i]$ into the query token. Hence, the CSP is not required to run a for loop N times (i.e. for the whole dictionary). Apart from that, FFP is also parallellizable - each query can be reduced to locating ℓ independent hashes in $\mathcal{D}$. Thus, by distributing the load to p processors, we get a total cost of $O(\ell/p)$.

Storage Cost Concerning the three indexes, we get that the size of $\mathcal{D}$, which is also the biggest index, is $O(N)$. The two indices that are stored locally on the user's side (i.e. $\mathsf{No.Runs}[f]$ and $\mathsf{No.Inputs}[f]$) are $O(m)$ each, where $m = |\mathcal{F}|$. Assuming that $\mathcal{F}$ is an extremely rich family of functions such that $\mathcal{F} = 1000$, and knowing that the average size of an integer is about 4 bytes, and the unique identifier of each function (i.e. the function description) is approximately 10 bytes, the total size required for the three indexes is: $1 \times 10^3 \times (10+4+10+4) = 1 \times 10^3 \times 28 = 28 \times 10^3\text{B} = 28\text{KB}$. Hence, we conclude the user, can easily store the two indexes even on a smartphone.

8.2 Experimental Results

Our experiments mainly focused on analyzing the FFP's performance. To do so, we implemented FFP in Python 3.9.4 using the `pandas`, `hashlib` and `numpy` libraries. To test the FFP's overall performance we used function families of different sizes, where each function had a different number of possible inputs. Our experiments focused on three main aspects: *(1)* Indexing, *(2)* Retrieving the inputs of a specific function and *(3)* Updating all the indexes by adding one new function in the function family. The indexes were implemented as `pandas` dataframes that are very common in data analysis. Additionally, as we wanted to evaluate the performance of FFP under realistic conditions, we decided to run our experiments on a commodity laptop. All experiments were executed on a Lenovo T470p with 2.81 GHz Intel Core i7 and 32GB RAM running Windows 10, 64-bit. The reason for measuring the FFP's performance on such a machine and not on a powerful desktop – as is the common practice – was that in a practical scenario, the most demanding processes (e.g. the creation of the dictionaries) take place on a user's machine. Hence, conducting the experiments on a powerful machine would have resulted in a set of non-realistic measurements. Each experiment was executed 100 times, and the average time was calculated.

Our Dataset: As already mentioned throughout the paper, FE schemes currently suffer from the absence of rich function families that could support a large number of different functions. However, to test the efficiency of our construction under the most demanding cases, we assumed the existence of such families and took it a step further by including in our experiments hypothetical

function families supporting up to 10,000 functions. Since FFP can work with any message-private MIFE scheme, measuring the encryption and decryption times is *not* applicable in our case. To this end, in our evaluation, we treat the function inputs as already encrypted elements of a large finite field.

Indexing: The first part of our experiments consisted of measuring the FFP.Setup time. During this phase, the data owner creates two local indexes (i.e. No.Inputs and No.Runs) and a dictionary $\mathcal{D}$ that will be outsourced to the CSP. Our datasets consisted of function families of different sizes, varying from a function family with 10 functions to function families with up to 10,000 different functions. All functions had an arbitrary number of inputs between 10 and 15,000. The functions were implemented as `{key: value}` dictionaries of the form {`function_id: list_of_inputs`}.

1. **Local Indexes** Generating the local indexes was a straightforward process executed through exploiting the structure of python dictionaries. Concerning the No.Inputs index, as for each function we had a {`function_id: list_of_inputs`} dictionary, we simply had to find the length of each `list_of_inputs` value. Moreover, for the No.Runs index, considering this was the setup phase, we only had to fill it with zeros. Table 2a illustrates the times required to generate the local indexes, in relation with the size of the function family (i.e. number of functions).

Table 2: Dataset Sizes and Setup Times

Size of Function Family	Time
10	0.0034s
100	0.032s
1,000	0.33s
5,000	1.7993s
10,000	3.66s

(a) Time required for the generation of the local indexes

Number of Functions	Number of Pairs	Time
10	49,127	0.02254s
100	499,819	0.1985s
1,000	4,993,007	2.0178
5,000	24,982,443	11.7846
10,000	50,036,227	28.2277

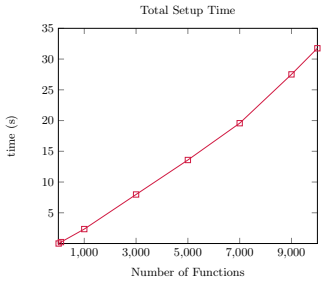
(b) Time required for the generation of the dictionary $\mathcal{D}$

2. **Remote Index ($\mathcal{D}$)**: Generating $\mathcal{D}$ was by far the most demanding process of as for each {`function_id: list_of_inputs`} dictionary we had to:
   - Retrieve the No.Inputs and No.Runs values corresponding to `function_id` and then mask `function_id` as shown in algorithm 1 to generate mf, and
   - For each possible input of a function x_i create a pair (mf, x_i) and add it into $\mathcal{D}$.

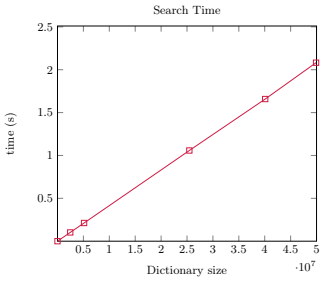
   The total number of (mf, x_i) pairs, increased dramatically as we increased the size of the function family. In particular, for a function family consisting of 10 functions, we ended up with 49,127 pairs and the time to generate $\mathcal{D}$ was measured at 0.02254s. In contrast, for a function family consisting of 10,000 functions, the total number of pairs was 50,036,227 and the required

time to generate $\mathcal{D}$ was 28.227s. Table 2b illustrates the number of pairs and the required time to generate $\mathcal{D}$ for different function families. Additionally, in figure 1b, we visualize the total time required for the successful completion of the FFP.Setup.

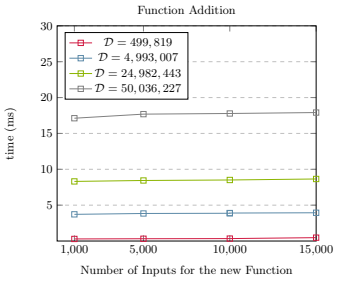
Query: In this part of the experiments we measured the time needed to complete a query over the dictionary $\mathcal{D}$. In our implementation, the query time was calculated as the sum of the time required to generate a token, find the corresponding matches at the database and update the indexes. The time required for the generation of the query token is independent of both the size of the function family and the size of the dictionary $\mathcal{D}$. On average, the time needed to generate the query token was measured at $8ms$. It needs to be noted that regarding the retrieval process, the actual process is equivalent to locating rows in the dataframe. More precisely, finding the inputs for a specific function over a set of 10 distinct functions and 49,127 rows required 0.0029 sec on average while retrieving the inputs for a specific function over a set of 10,000 functions and 50,036,227 rows took 2,082s. Figure 1b illustrates our results.



(a) Total Search Time



(b) Correlation between search time and size of $\mathcal{D}$



(c) Adding a new function with different number of inputs

New Function Addition: In the last part of our experiments we focused on measuring the time required to add a new function to the function family. This process required updating both the local indexes and $\mathcal{D}$. As already mentioned,

all the indexes were implemented as pandas dataframes and all functions were read as {`function_id: list_of_inputs`} dictionaries. To this end, to update all the indexes we had to: *(1)* Retrieve both the local indexes and $\mathcal{D}$, *(2)* Build two dataframes out of the {`function_id: list_of_inputs`} dictionary, just like in the Setup phase, and *(3)* Merge the new dataframes with the existing ones. To get a better picture of the computational time required for this process, we ran this experiment for all the different size dictionaries $\mathcal{D}$ that we generated in the Setup phase. Another variable in this experiment was the number of inputs for the new function. More precisely, we measured the time required to add a function whose number of inputs varied from 1,000 to 15,000. What we found was that the total time required to add a new function was almost independent of the function's number of inputs, as on average the number of inputs affected the running time by a total of 0,1ms. Conversely, the running time was drastically affected by the size of the existing dictionary $\mathcal{D}$, as it varied from 0.45ms for a dictionary the size of 499,819 to almost 17.5ms for a dictionary the size of 50,036,227. Figure 1c visualises our results.

9 Limitations and Future Directions

TEE Dependence The first FFP limitation is that currently, we assume the existence of a fully trusted authority to perform functional decryption. This is not a novel concept in FE, as similar approaches appear in state-of-the-art literature [21], where authors rely on Intel SGX for this task. In contrast to [21], where the TEE decrypts the ciphertexts and then applies the function on the plaintexts, we feel that our work is more consistent with the traditional definitions of FE, where the evaluation of the function occurs directly on the ciphertexts. Nevertheless, we acknowledge that relying on a TEE is not a very elegant solution and to this end, we plan to entirely stop using it in our constructions in the future. To achieve this, we will explore the following two solutions: *(1)* The first solution is rather naive and involves a more active participation of the data owner. More precisely, after the CSP retrieves all the inputs for the function, the data owner wishes to run, instead of sending them to SC, it forwards them back to the data owner. The data owner can then run the decryption algorithm locally and acquire the wanted results. *(2)* The second solution is to use a decentralized storage network (DSN), such as a blockchain in the place not only of the TEE but also the CSP. In such a scenario, the ciphertexts will be stored in a decentralized manner and multiple authorities could work together towards the retrieval of the inputs and the evaluation of the function, without being aware of one another.

Number of Users The second FFP limitation revolves around the fact that even though it works with any MIFE scheme, the main scenario is built around the single-client setting. Extending it to the multi-client setting, normally requires the existence of a fully trusted central authority responsible for the generation and distribution of functional decryption keys. However, as we already

mentioned, our goal is to transform FFP so that the TEE assumption is ultimately eliminated. We believe the way forward is in using a decentralized approach. This way, the need for a fully trusted entity will be obviated by shifting the task of generating functional keys to the clients themselves.

10 Conclusion

In this work, we presented a novel threat model for Functional Encryption by identifying limitations of current state-of-the art approaches. By using this adversarial model, we were able to quantify the leakage during the execution of a Functional Encryption scheme – a problem that has so far been overlooked in the current literature. We firmly believe that our results are very important and pave the way towards designing more secure and robust FE schemes. Moreover, based on our theoretical formulation, we designed FFP – a framework that yields a function-private FE scheme, based on any message-private one. We believe our approach can be seen as a valuable contribution to the the fields of cryptography and security as we showed how to extend existing techniques, by addressing a series of limitations and assuming stronger threat models.

References

1.
2. Abdalla, M., Bourse, F., De Caro, A., Pointcheval, D.: Simple functional encryption schemes for inner products. In: IACR International Workshop on Public Key Cryptography. pp. 733–751. Springer (2015)
3. Abdalla, M., Catalano, D., Fiore, D., Gay, R., Ursu, B.: Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 597–627. Springer International Publishing, Cham (2018)
4. Abdalla, M., Gay, R., Raykova, M., Wee, H.: Multi-input inner-product functional encryption from pairings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2017)
5. Agrawal, S., Agrawal, S., Badrinarayanan, S., Kumarasubramanian, A., Prabhakaran, M., Sahai, A.: Function private functional encryption and property preserving encryption: New definitions and positive results. IACR Cryptol. ePrint Arch. **2013**, 744 (2013)
6. Ananth, P., Jain, A., Sahai, A.: Robust transforming combiners from indistinguishability obfuscation to functional encryption. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 91–121. Springer (2017)
7. Bakas, A., Michalas, A.: Multi-input functional encryption: Efficient applications from symmetric primitives. In: Wang, G., Ko, R.K.L., Bhuiyan, M.Z.A., Pan, Y. (eds.) 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020, Guangzhou, China, December 29, 2020 - January 1, 2021. pp. 1105–1112. IEEE (2020). https://doi.org/10.1109/TrustCom50675.2020.00146, `https://doi.org/10.1109/TrustCom50675.2020.00146`

8. Bakas, A., Michalas, A.: Nowhere to leak: A multi-client forward and backward private symmetric searchable encryption scheme. In: Barker, K., Ghazinour, K. (eds.) Data and Applications Security and Privacy XXXV. pp. 84–95. Springer International Publishing, Cham (2021)
9. Bakas, A., Michalas, A., Dimitriou, T.: Private lives matter: A differential private functional encryption scheme. In: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy. p. 300–311. CODASPY '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3508398.3511514, https://doi.org/10.1145/3508398.3511514
10. Barbosa, M., Catalano, D., Soleimanian, A., Warinschi, B.: Efficient function-hiding functional encryption: From inner-products to orthogonality. In: Cryptographers' Track at the RSA Conference. pp. 127–148. Springer (2019)
11. Bishop, A., Jain, A., Kowalczyk, L.: Function-hiding inner product encryption. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 470–491. Springer (2015)
12. Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. In: Annual international cryptology conference. pp. 213–229. Springer (2001)
13. Boneh, D., Raghunathan, A., Segev, G.: Function-private identity-based encryption: Hiding the function in functional encryption. In: Annual Cryptology Conference. pp. 461–478. Springer (2013)
14. Boneh, D., Raghunathan, A., Segev, G.: Function-private subspace-membership encryption and its applications. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 255–275. Springer (2013)
15. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Theory of Cryptography Conference. pp. 253–273. Springer (2011)
16. Brakerski, Z., Komargodski, I., Segev, G.: Multi-input functional encryption in the private-key setting: Stronger security from weaker assumptions. Journal of Cryptology **31**(2), 434–520 (2018)
17. Brakerski, Z., Segev, G.: Function-private functional encryption in the private-key setting. Journal of Cryptology **31**(1), 202–225 (2018)
18. Carmer, B., Malozemoff, A.J., Raykova, M.: 5gen-c: multi-input functional encryption and program obfuscation for arithmetic circuits. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 747–764 (2017)
19. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptol. ePrint Arch. **2016**(86), 1–118 (2016)
20. Dowsley, R., Michalas, A., Nagel, M., Paladi, N.: A survey on design and implementation of protected searchable data in the cloud. Computer Science Review (2017). https://doi.org/https://doi.org/10.1016/j.cosrev.2017.08.001, http://www.sciencedirect.com/science/article/pii/S1574013716302167
21. Fisch, B., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: Iron: functional encryption using intel sgx. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 765–782 (2017)
22. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. SIAM Journal on Computing **45**(3), 882–929 (2016)
23. Goldwasser, S., Gordon, S.D., Goyal, V., Jain, A., Katz, J., Liu, F.H., Sahai, A., Shi, E., Zhou, H.S.: Multi-input functional encryption. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 578–602. Springer (2014)

24. Kim, S., Lewi, K., Mandal, A., Montgomery, H., Roy, A., Wu, D.J.: Function-hiding inner product encryption is practical. In: International Conference on Security and Cryptography for Networks. pp. 544–562. Springer (2018)
25. Komargodski, I., Segev, G.: From minicrypt to obfustopia via private-key functional encryption. Journal of Cryptology **33**(2), 406–458 (2020)
26. Komargodski, I., Segev, G., Yogev, E.: Functional encryption for randomized functionalities in the private-key setting from minimal assumptions. Journal of Cryptology **31**(1), 60–100 (2018)
27. Sahai, A., Seyalioglu, H.: Worry-free encryption: functional encryption with public keys. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 463–472 (2010)
28. Sans, E.D., Gay, R., Pointcheval, D.: Reading in the dark: Classifying encrypted digits with functional encryption. IACR Cryptology ePrint Archive **2018**, 206 (2018)
29. Tomida, J.: Tightly secure inner product functional encryption: Multi-input and function-hiding constructions. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 459–488. Springer (2019)
30. Waters, B.: A punctured programming approach to adaptively secure functional encryption. In: Annual Cryptology Conference. pp. 678–697. Springer (2015)
31. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: 2015 IEEE Symposium on Security and Privacy. pp. 640–656. IEEE (2015)
32. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th {USENIX} Security Symposium ({USENIX} Security 16). pp. 707–720 (2016)