A compiler for multi-key homomorphic signatures for Turing machines*

Somayeh Dolatnezhad Samarin[1], Dario Fiore[2], Daniele Venturi[3], and Morteza Amini[1]

[1]Sharif University of Technology, Tehran, Iran
[2]IMDEA Software Institute, Madrid, Spain
[3]Sapienza University of Rome, Rome, Italy

Abstract

At SCN 2018, Fiore and Pagnin proposed a generic compiler (called "Matrioska") allowing to transform sufficiently expressive single-key homomorphic signatures (SKHSs) into multi-key homomorphic signatures (MKHSs) under falsifiable assumptions in the standard model. Matrioska is designed for homomorphic signatures that support programs represented as circuits. The MKHS schemes obtained through Matrioska support the evaluation and verification of arbitrary circuits over data signed from multiple users, but they require the underlying SKHS scheme to work with circuits whose size is *exponential* in the number of users, and thus can only support a constant number of users.

In this work, we propose a new generic compiler to convert an SKHS scheme into an MKHS scheme. Our compiler is a generalization of Matrioska for homomorphic signatures that support programs *in any model of computation*. When instantiated with SKHS for circuits, we recover the Matrioska compiler of Fiore and Pagnin. As an additional contribution, we show how to instantiate our generic compiler in the Turing Machines (TM) model and argue that this instantiation allows to overcome some limitations of Matrioska:

- First, the MKHS we obtain require the underlying SKHS to support TMs whose size depends only *linearly* in the number of users.
- Second, when instantiated with an SKHS with succinctness $\mathsf{poly}(\lambda)$ and fast enough verification time, e.g., $S \cdot \log T + n \cdot \mathsf{poly}(\lambda)$ or $T + n \cdot \mathsf{poly}(\lambda)$ (where T, S, and n are the running time, description size, and input length of the program to verify, respectively), our compiler yields an MKHS in which the time complexity of both the prover and the verifier remains $\mathsf{poly}(\lambda)$ even if executed on programs with inputs from $\mathsf{poly}(\lambda)$ users.

While we leave constructing an SKHS with these efficiency properties as an open problem, we make one step towards this goal by proposing an SKHS scheme with verification time $\mathsf{poly}(\lambda) \cdot T$ under falsifiable assumptions in the standard model.

1 Introduction

Consider a user Alice who outsources storage of her data x to a powerful server, and let Bob be another user (also referred here as client) who wants to evaluate a public function f on Alice's data

x. Since x may be very large (and f may be expensive) Bob can delegate this computation to the server which evaluates $y = f(x)$ and returns y to Bob. How can Bob be convinced that the server's computation was done correctly, when receiving only a small amount of information (i.e., much less than the size of x)?

Homomorphic signatures [7] provide a generic solution to this problem. In homomorphic signatures, Alice has a private signing key sk corresponding to a public verification key pk that is known to anyone, including Bob and the server. Alice sends to the server x along with a signature σ on x. The server can now compute y as before, and homomorphically generate a signature σ^* proving that indeed $y = f(x)$. A distinguishing property of homomorphic signatures is that the size of σ^* is significantly smaller than that of x, i.e., logarithmic or constant in $|x|$. Note that without such requirement homomorphic signatures could be trivially realized from regular digital signatures as the server could send to Bob x and Alice's signature on it. If the computational efficiency of verifying σ^* with respect to f is also a concern (namely, it is too expensive for Bob), some works introduced the notion of efficient verification. The idea is that Bob can do a one-time preprocessing for the function f, and later verify any signatures for f's outputs in constant time.

Multi-key homomorphic signatures (MKHSs) [15] are a generalization of homomorphic signatures (from hereon referred to as "single-key homomorphic signatures", SKHS, for distinction) to $t \geq 1$ users. In MKHS, each user has a pair of keys (pk_i, sk_i) and signs its own input x_i obtaining a signature σ_i that is outsourced to the server (along with the corresponding input). The server now computes $y = f(x_1, \ldots, x_t)$ and homomorphically evaluates a signature σ^* proving that y was computed correctly. Verification of σ^* now requires the public keys of all the clients.

MKHS were introduced by Fiore *et al.* [15] who proposed a construction based on lattices. In 2018, Fiore and Pagnin [16] proposed a generic compiler (called Matrioska) for turning SKHSs into MKHSs. Using their transform, each user can sign its own input using the signing algorithm of the underlying SKHS. Matrioska exploits the homomorphic property of the SKHS to combine the signatures from different clients in t steps. The length of the final signature σ^* is $t \cdot \ell$, where ℓ is the signature length in the underlying SKHS.

Although the result of [16] establishes a general connection between SKHSs and MKHSs, this result is limited to the case when the number of users is a small constant. The reason of this limitation is that, in order to create an MKHS scheme that supports the evaluation of a circuit of size s, the Matrioska compiler needs to start from an SKHS that supports a circuit of size $s^{c^{t-1}}$, where c is some constant that depends on the SKHS scheme.

This double-exponential dependence on t stems from the compiler's approach of [16] in which one builds t circuits such that, roughly speaking, the circuit at step i takes as input the description of the circuit built in the previous step $i-1$. So, by assuming that each circuit has size polynomial in its input length, the growth is double-exponential.

1.1 Our results

The first contribution of our work is the proposal of a new generic compiler for turning an SKHS into an MKHS and that supports the evaluation and verification of programs represented in *any model of computation*.

We designed our compiler by abstracting away the compiler of [16] (that was designed to work specifically for programs represented as circuits) in order to support general computational models. Such abstraction allows us to separate the steps of the compiler that rely on the properties of the SKHS from those steps that instead solely depend on the given computational model. We believe

this separation also offers a better explanation of the Matrioska compiler. A crucial building block for the steps unrelated to the SKHS scheme is an algorithm, called Mask, that on input the description of a function F and a portion of the input x (the suffix x'' that we want to fix in the description of the function), outputs the description of another function F' that is the partial application of F on x'', i.e., s.t. for any $x = (x', x'')$, $F'(x') = F(x', x'')$.

A bit more precisely, to be used in our compiler, this Mask algorithm must satisfy some other properties related to efficiently locating the fixed input in F' description and to recursively applying Mask (due to the high technicality of these properties we refer to Section 4.1 for more details).

Therefore, to instantiate our compiler in a given model of computation one needs to plug in two main ingredients: an SKHS and the implementation of Mask, both for programs described in the given model.

Since our compiler is abstract we cannot provide a concrete efficiency analysis of the MKHS it produces. What we provide generically, though, is a framework to conduct such analysis based on the following parameters:

- the efficiency of the SKHS verification algorithm, expressed as two functions $T_V(\cdot)$ and $S_V(\cdot)$ that determine the running time and the description size of the verification algorithm in terms of the running time T and size S of the program to verify;
- the efficiency of the Mask function, i.e., the size and running time of the function with hardcoded inputs that it returns.

The second parameter, the efficiency of Mask, is specific to the computational model as its implementation (and complexity) depends on how programs are represented. The first parameter, the efficiency of SKHS verification, is specific to the SKHS scheme one starts from. To test our compiler we consider five representative cases that span from a realistic one (i.e., the one achieved by the SKHS of [21] where verification would take time $p(\lambda)T \log T$) to a nearly optimal one, where the verification time is $S + n \cdot \mathsf{poly}(\lambda)$ (where T, S, and n are the running time, description size, and input length of the program to verify, respectively).

Next, we show that the Matrioska compiler of [16] can be seen as an instantiation of our compiler in the circuits model. For this case we revisit the efficiency analysis of [16] that estimate the size of the circuits supported by the SKHS scheme in order to be used to create an MKHS for t users. We show that, even under the most favorable assumption about the SKHS verification, i.e., $S + n \cdot \mathsf{poly}(\lambda)$, the SKHS must support circuits of size *at least* exponential in t. Such lower bound somehow shows the limits of applying this compiler to the circuits model.

Our second main contribution is then to propose an instantiation of our compiler for programs represented as (multi-tape) Turing machines. In particular, our technical contribution is the design of the Mask algorithm for programs represented as TMs thanks to which we can overcome the aforementioned limitations of the circuits model (and of the Matrioska compiler). The first advantage of our compiler for TMs is that, independently of the efficiency of the SKHS verification, we obtain an MKHS scheme that supports the multi-key evaluation of a TM of size S for t users, starting from an SKHS scheme that supports TMs of size, roughly, $S + t \cdot S_v$, where S_v is the (fixed) size of the TM expressing the SKHS scheme's verification algorithm. Namely, with respect to the size of the TMs, *we obtain only a linear dependence on the number of users.* In contrast, in the original Matrioska transform, the circuits size supported by the SKHS scheme must be *exponentially* smaller than that supported in the multi-key evaluation. The second advantage is that, by assuming the underlying

SKHS to have verification times ($T_V(T, S, n, \lambda) = T + np(\lambda)$, $T_V(T, S, n, \lambda) = S \log T + np(\lambda)$, or $T_V(T, S, n, \lambda) = S + np(\lambda)$), we obtain an MKHS in which the underlying SKHS is executed on programs whose running time depends only linearly on t. This means that our compiler yields an MKHS that can support up to $\mathsf{poly}(\lambda)$ number of users. In contrast, in the circuit model, under the same assumption about T_V, the compiler yields an MKHS that can support constant number of users. We defer the reader to Section 6.3 for the detailed analysis.

Towards efficiently instantiating our compiler To the best of our knowledge, no existing SKHS directly supports Turing machines, and thus our compiler needs to take into account the overhead required to represent Turing machines as circuits. If we consider the state-of-the-art SKHS [21], its verification time for a TM running in time at most T is $\mathsf{poly}(\lambda) \cdot T \log(T)$, which does not fit the most efficient cases of our compiler mentioned earlier (i.e., those allowing to support a super-constant number of users).

As the last contribution of this paper, we make progress towards closing this gap. In particular, we show how to construct an efficiently verifiable SKHS with verification time $T^{O(\frac{1}{\log_2(\log_\lambda(T))})} + n \cdot \mathsf{poly}(\lambda)$ from any SKHS with verification time T. The latter is achieved leveraging so-called non-interactive delegation systems, which exist under falsifiable assumptions in the standard model [23]. The only drawback is that the signature length is $T^{O(\frac{1}{\log_2(\log_\lambda(T))})}$, and thus still depends on T. When using the SKHS scheme of [21] in the instantiation, we can obtain a single-key scheme with verification time $\mathsf{poly}(\lambda)T^{(\mathrm{p})}$ that for the sake of our compiler needs to support TMs running up to $\mathsf{poly}(\lambda)^t T^{(\mathrm{p})}$ time. We leave it as an open problem to design a delegation scheme with better succinctness, which would directly allow our MKHS compiler to have a polynomial verification time, and thus support an arbitrary polynomial number of users.

1.2 Paper organization

In Section 2, we review known constructions of homomorphic signatures. We recall a few standard preliminaries in Section 3. The general definition of the compiler is described and analyzed in Section 4. We recall the original Matrioska compiler in Section 5 and do an efficiency analysis of this compiler. In Section 6, we proposed our TM-based compiler and analyzed its efficiency. Finally, in Section 7, we propose a generic transform for improving the verification time of SKHSs. We conclude our paper in Section 8.

2 Related work

Homomorphic signatures were introduced by Johnson et al. [22], and the first scheme for computing linear functions over signed vectors was proposed by Boneh et al. [6]. Several works then proposed constructions of linearly-homomorphic signatures [1, 8, 3, 11, 12, 18, 4, 25, 9, 14, 10]. Only a few works propose constructions that support more expressive functions such as polynomials [7, 13] and circuits of bounded polynomial depth [21].

In [15], Fiore *et al.* introduced multi-key homomorphic signatures and showed a construction for functions represented as circuits with bounded depth under standard lattice-based assumptions. Compared to the scheme in [15], the MKHS schemes obtained from our compiler perform worse, as [15] can tolerate a polynomial number of users while keeping the complexity of the scheme polynomial-time. However we stress that [15] builds a scheme based on specific algebraic techniques,

whereas the goal of our work is to establish a general result that works for *any* SKHS. Lai *et al.* [24] show how to construct MKHSs using SNARKs and standard digital signatures; however, SNARKS inherently require non-falsifiable assumptions [20]. More recently, Schabhüser *et al.* [26] and Aranha *et al.* [2] construct MKHSs supporting linear functions using bilinear maps.

3 Preliminaries

Notation. Let λ be the security parameter, $\mathcal{M}$ be a message space, $\mathsf{poly}(\lambda)$ be a polynomial function based on λ, and let $[n] := \{1, \ldots, n\}$. The notation $s \xleftarrow{\$} S$ denotes uniformly sampling a value s from a set S. A function $negl(\lambda)$ said to be negligible in λ, if for every polynomial p, there exists an integer N such that for all integers $n > N$, $negl(\lambda) < \frac{1}{p(n)}$. If A is a probabilistic algorithm (i.e., it uses random coins), $y \leftarrow A(\cdot)$ denotes assigning the output of the execution of A to the variable y. We use the notation $O(1)$ for a constant value and $p(\lambda)$ for a fixed polynomial based on the security parameter λ.

3.1 Homomorphic signature schemes

In this section, we first recall the notion of labeled programs introduced by Gennaro and Wichs [19] and extended by Fiore et al. [15] for programs with inputs from more than one user. Then, we review the definitions of SKHS and MKHS schemes and the correctness, succinctness, and security properties of them.

Labeled programs The input program in most single-key and multi-key homomorphic signature schemes is modeled as a labeled program. A labeled program $\mathcal{P} = (f, (\ell_1, ..., \ell_n))$ consists of an n-variate function $f : \mathcal{M}^n \to \mathcal{M}$ and a set of labels $\ell_1, ..., \ell_n \in \{0,1\}^*$. Labeled programs $\mathcal{P}_1$,...,$\mathcal{P}_N$ can be composed using a function $G : \mathcal{M}^N \to \mathcal{M}$. The function G evaluates on the outputs of $\mathcal{P}_1, ..., \mathcal{P}_N$. The inputs of the composed program $\mathcal{P}^* = G(\mathcal{P}_1, ..., \mathcal{P}_N)$ are all distinct inputs of the labeled programs $\mathcal{P}_1, ..., \mathcal{P}_N$ (the inputs with the same labels are grouped together). For multi-key homomorphic signatures [15], the identity of the user (i.e. id) are added to the labels such that $\ell = (id, \tau)$ where τ is a tag. Actually, τ is a string to determine a data item in a set of inputs generated by the user with identity id.

Multi-labeled programs [5] A multi-labeled program $\mathcal{P}_\Delta$ is a pair $(\mathcal{P}, \Delta)$, where $\mathcal{P}$ is a labeled program and $\Delta \in \{0,1\}^*$ is a dataset identifier. Multi-labeled programs $\mathcal{P}_{1,\Delta}, \mathcal{P}_{2,\Delta}, ..., \mathcal{P}_{N,\Delta}$ with the same Δ, can also be composed using a function $G : \mathcal{M}^N \to \mathcal{M}$ as $\mathcal{P}^*_\Delta = G(\mathcal{P}_1, ..., \mathcal{P}_N)$

In SKHS schemes, labels in a labeled program are tags used to specify on which inputs, among a set of data items, the program is to be executed.

Definition 1 (Multi-key homomorphic signature scheme [15])**.** *A multi-key homomorphic signature scheme is a tuple of the following five probabilistic polynomial time (PPT) algorithms* $\Pi_{\mathsf{MH}} = (\mathsf{MH.Setup}, \mathsf{MH.KeyGen}, \mathsf{MH.Sign}, \mathsf{MH.Eval}, \mathsf{MH.Verif})$.

- $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda)$*: given the security parameter* λ*, this algorithm outputs the public parameter* pp *which is the default input of other algorithms. This parameter describes a message space* $\mathcal{M}$*, a label space* $\mathcal{L} = \mathcal{ID} \times \mathcal{T}$*, where* $\mathcal{ID}$ *is an identity space and* $\mathcal{T}$ *is a tag space, a signature space* $\mathcal{Y}$*, and a set of admissible functions* $F : \mathcal{M}^n \to \mathcal{M}$*.*

- $(sk, pk) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})$*: given the public parameter pp, this algorithm outputs a secret key sk and a public key pk.*

- $\sigma \leftarrow \mathsf{MH.Sign}(sk, \Delta, \mathrm{m}, \ell = (id, \tau))$*: given the secret key sk, a dataset identifier* $\Delta \in \{0,1\}^\lambda$*, a message* $\mathrm{m} \in \mathcal{M}$*, and a label of the message* $\ell \in \mathcal{L}$*, this algorithm outputs a signature* $\sigma \in \mathcal{Y}$*.*

- $\sigma' \leftarrow \mathsf{MH.Eval}(\mathcal{P}, \Delta, \{\sigma_i, pk_{id_i}\}_{i=1}^n)$*: given a labeled program* $\mathcal{P}$*, a dataset identifier* $\Delta \in \{0,1\}^\lambda$*, and a set of public key and signature pairs* $\{(\sigma_i, pk_{id_i})\}_{i=1}^n$*, this algorithm outputs an authenticator* σ' *that is supposed to vouch for the correctness of the result message* $\mathrm{m} = f(\mathrm{m}_1, ..., \mathrm{m}_n)$*.*

- $b \leftarrow \mathsf{MH.Verif}(\mathcal{P}, \Delta, \{pk_{id}\}_{id \in \mathcal{P}}, \sigma', \mathrm{m})$*: given a labeled program* $\mathcal{P}$*, a dataset identifier* $\Delta \in \{0,1\}^\lambda$*, a set of public keys for identities contributed in* $\mathcal{P}$*, an authenticator* σ'*, and a message* $\mathrm{m} \in \mathcal{M}$*, this algorithm outputs* $b = 1$ *for accepting or* $b = 0$ *for rejecting the result.*

MKHS correctness An MKHS scheme is correct if it has the authentication correctness and evaluation correctness properties defined as follows.

- **Authentication correctness**: Let $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda)$, and let $(sk_{id}, pk_{id}) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})$ be a key pair for any user with identity $id \in \mathcal{ID}$. For any $\mathrm{m} \in \mathcal{M}$, any dataset identifier $\Delta \in \{0,1\}^\lambda$, and any $\ell = (id, \tau) \in \mathcal{L}$, if σ is the output of $\mathsf{MH.Sign}(sk_{id}, \Delta, \mathrm{m}, \ell)$, then we have $\mathsf{MH.Verif}(\mathcal{I}, \Delta, pk_{id}, \sigma, \mathrm{m}) = 1$, where $\mathcal{I} = (I, \ell)$ is the labeled program for the identity function such that $I(\mathrm{m}) = \mathrm{m}$.

- **Evaluation correctness**: Let $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda)$, and let $\{(sk_{id}, pk_{id}) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})\}_{id \in \hat{\mathcal{ID}}}$ be a set of key pairs for a set of users with identifier $id \in \hat{\mathcal{ID}}$ where $\hat{\mathcal{ID}} \subseteq \mathcal{ID}$. Let $G : \mathcal{M}^\omega \rightarrow \mathcal{M}$ be a function for composing labeled programs. For any triples $\{(\mathcal{P}_i, \mathrm{m}_i, \sigma_i)\}_{i=1}^\omega$ and a fixed dataset identifier Δ, if for each i, $\mathsf{HS.Verif}(\mathcal{P}_i, \Delta, \{pk_{id}\}_{id \in \mathcal{P}_i}, \sigma_i, \mathrm{m}_i) = 1$, then we have $\mathsf{HS.Verif}\big(\mathcal{P}^*, \Delta, \{pk_{id}\}_{id \in \mathcal{P}^*}, \sigma^*, \mathrm{m}^*\big) = 1$, where $\mathcal{P}^* = G(\mathcal{P}_1, ..., \mathcal{P}_\omega)$, $\mathrm{m}^* = G(\mathrm{m}_1, ..., \mathrm{m}_\omega)$ and $\sigma^* = \mathsf{HS.Eval}(G, \Delta, \{\sigma_i, \{pk_{id}\}_{id \in \mathcal{P}_i}\}_{i=1}^\omega)$.

MKHS succinctness Let $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda)$, let $\mathcal{P} = (f, \ell_1, ..., \ell_n)$ be a labeled program with $\ell_i = (id_i, \tau_i)$, let $\{(sk_{id}, pk_{id}) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})\}_{id \in \{id_1, ..., id_n\}}$ be a set of key pairs, and let $\{\sigma_i \leftarrow \mathsf{MH.Sign}(sk_{id_i}, \Delta, \mathrm{m}_i, \ell_i = (id_i, \tau_i))\}_{i=1}^n$ be a set of signatures. We say that an MKHS scheme has the succinctness property if and only if the size of the authenticator $\sigma' \leftarrow \mathsf{MH.Eval}(\mathcal{P}, \Delta, \{\sigma_i, pk_{id}\}_{id \in \mathcal{P}})$ logarithmically depends on n, but possibly linearly in t. Namely, there is a fixed polynomial p such that $|\sigma'| = p(\lambda, t, \log(n))$.

MKHS security In the security model defined by Fiore et al. [15], in addition to asking for signatures, the adversary can also corrupt signers. Since each user has its own secret and public keys, corrupting a user doesn't violate the integrity of computations on messages signed by other users. Let $\Pi_{\mathsf{MH}} = (\mathsf{MH.Setup}, \mathsf{MH.KeyGen}, \mathsf{MH.Sign}, \mathsf{MH.Eval}, \mathsf{MH.Verif})$ be an MKHS scheme and A be a probabilistic polynomial-time adversary. To define the security of the scheme, we define a game $exp_{A,\Pi_{\mathsf{MH}}}^{MKHS}(\lambda)$ between the adversary A and the challenger C as follows:

- **Setup**: We assume that the adversary A knows the security parameter λ. The challenger C initializes $L_{corr} = \emptyset$ as the of corrupted identities, runs $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda)$ and sends the public parameter pp to A.

- **Signing Queries**: A is given the oracle access to the $\mathsf{MH.Sign}$ algorithm (O_{Sign}) and can adaptively send requests of the form $(\Delta, \ell = (id, \tau), \mathrm{m})$ to O_{Sign} where $\Delta \in \{0,1\}^{\lambda}$, $\mathrm{m} \in \mathcal{M}$, $id \in \mathcal{ID}$ and $\tau \in \mathcal{T}$. When A sends his/her query, C examines the following conditions and sends σ to A:
  - If L_{Δ} does not exist (i.e., it is the first query with dataset Δ) C creates $L_{\Delta} \leftarrow \emptyset$.
  - If no signing query with label $\ell = (id, \cdot)$ or corruption query on id was ever issued, C generates and stores $(sk_{id}, pk_{id}) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})$.
  - If L_{Δ} already contains a pair (ℓ, m') for some message m', C ignores the query.
  - If L_{Δ} does *not* contain any pair $(\ell, \cdot)$, C computes $\sigma \leftarrow \mathsf{HS.Sign}(sk, \Delta, \mathrm{m}, \ell)$, and adds (τ, m) to L_{Δ}.
- **Corruption Queries**: In corruption queries, A is given oracle access to $\mathsf{MH.KeyGen}$ algorithm (O_{KeyGen}) and can adaptively send queries of the form $q = id$ to O_{KeyGen}. During the game, C initiates an empty list L_{corr} of corrupted users. When A sends a corruption query $q = id$ such that $id \in \mathcal{ID}$ to C, C examines the following conditions and sends (sk_{id}, pk_{id}) to A:
  - If A request id for the first time, C adds id to L_{corr} and returns (sk_{id}, pk_{id}) (if a key pair for id was not generated before, it generates it in this step, $(sk_{id}, pk_{id}) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})$).
  - If A request an identity which was requested before (i.e., $id \in L_{corr}$), C sends to A the previously computed pair (sk_{id}, pk_{id}).
- **Forgery**: Finally, A outputs a tuple $(\mathcal{P}^*, \Delta^*, \mathrm{m}^*, \sigma'^*)$ where $\mathcal{P}^* = (f^*, \ell_1^*, ..., \ell_{\omega}^*)$. The adversary A wins the game, if and only if $\mathsf{MH.Verif}(\mathcal{P}^*, \Delta^*, \{pk_{id}\}_{id \in \mathcal{P}^*_{\Delta^*}}, \sigma'^*, \mathrm{m}^*) = 1$, for all $id \in \mathcal{P}^*$, $id \notin L_{corr}$, and at least one of the following forgeries occurs:
  1. **Type-I forgery**: Δ^* is a new dataset which is not queried before.
  2. **Type-II forgery**: Δ^* is not a new dataset (Δ^* was queried to the signing oracle), for all $i \in [\omega]$, $(\tau_i^*, \mathrm{m}_i) \in L_{\Delta^*}$ and $\mathrm{m}^* \neq f^*(\mathrm{m}_1, ..., \mathrm{m}_{\omega})$.
  3. **Type-III forgery**: L_{Δ^*} exists and there exists at least an index $j \in [\omega]$ such that $(\tau_j^*, \cdot) \notin L_{\Delta^*}$. In other words, $(\ell_1^*, ..., \ell_{\omega}^*)$ contains at least one label $\ell_j = (id_j, \tau_j)$ which was never queried to the signing oracle and id_j is not a corrupted identity, namely $id_j \notin L_{corr}$.

We say that scheme Π_{MH} is secure if for all probabilistic polynomial-time adversaries A, there is a negligible function $negl(.)$ such that:

$$\Pr[\exp^{MKHS}_{A,\Pi_{\mathsf{MH}}}(\lambda) = 1] \leq negl(\lambda)$$

Non-adaptive corruption queries We remind the reader about a proposition shown in [15] to argue that, when corruption queries are made non adaptively, it is enough to prove security for adversaries that make no corruptions.

Proposition 1 ([15])**.** *MKHS is secure against adversaries that do not make corruption queries if and only if MKHS is secure against adversaries that make non-adaptive corruption queries.*

Remark 1. *An instantiation of the MKHS definition where only one key is generated and the same id is used for all labels covers the definition of the SKHS scheme as a special case.*

4 Our generic compiler from SKHS to MKHS

In this section we propose our generic compiler that converts a single-key homomorphic signature scheme into a multi-key homomorphic signature scheme. Our compiler generalizes the one proposed by Fiore et al. [16] (called "Matrioska") that works for homomorphic signature schemes for programs that are modeled as circuits. In contrast, our compiler works for schemes that supports programs defined in any model of computation (not limited to circuits).

4.1 Notation and building blocks

To define our compiler, we first introduce some notation and building blocks.

Notation. We denote by $\mathcal{M}$ the message space of the SKHS scheme. We denote a function with n inputs and m outputs by $F : \mathcal{M}^n \to \mathcal{M}^m$. By slightly abusing notation, we also use its name, namely F, to denote the description of the function, which is assumed to be a string in $\mathcal{M}^\ell$ for some integer length ℓ. For a function F we denote by $T(F)$ its running time, and by $S(F)$ (or $|F|$) the size of its description. Note that the description depends on the computational model at hand, e.g., it can be the description of a circuit, a Turing machine or a RAM machine.

Definition 2 (Equality function ($\mathbf{EQ^y}$))**.** *For a given $y \in \mathcal{M}$, the equality function $EQ^y(x)$ takes as input a value $x \in \mathcal{M}$ and is defined as follows:*

$$EQ^y(x) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases},$$

Definition 3 (Composition function ($Compose$))**.** *We assume the existence of a polynomial-time computable function Compose that on input the description of two functions F, G outputs the description of a function H that computes their sequential composition. More formally, on input two functions*

$$F : \mathcal{M}^n \to \mathcal{M}^m,\ G : \mathcal{M}^m \to \mathcal{M}^\ell,$$

we have $H \leftarrow Compose(F, G)$ such that $\forall x \in \mathcal{M}^n$

$$H(x) = G(F(x))$$

Masking function (Mask). This is a central building block of our compiler. Intuitively, Mask is a polynomial-time computable function that on input the description of a function F and a portion of an input string, it outputs the description of a function F' which computes the same as F but with the specified inputs already fixed. Namely, F' is a partial application of F. In addition to this basic functionality we assume a few more properties.

The first one is that one can specify a set of indices, among the input symbols that are fixed, so that for each of these indices Mask returns the position in the description of F' where the corresponding symbol was written. Essentially we can keep track of where the fixed symbols are in the description of the new function. This property is useful in the compiler where we provide F' as

an "almost" fixed input of the SKHS verification function, where "almost" is due to the fact that some of the inputs fixed in F' are not provided and thus we need to locate where they are.

The other property of Mask is more specific to our application in the compiler. Let us say there are t functions $G_1, \ldots, G_t$ and a function E_1 which encodes fixed inputs $(x_2, \ldots, x_t)$. Also, let us assume that one iteratively applies Mask on the function G_i and a fixed input that is the description of E_i except for the input value x_{i+1}. Notice that with such iterative applications, at the step $t-1$ one obtains a function E_t in which none of the original inputs $(x_2, \ldots, x_t)$ is fixed anymore. Then the last property of Mask requires that for such a scenario there exists an alternative algorithm Mask^* that can produce directly E_t without knowing $(x_2, \ldots, x_t)$.

We stress that in this section we only provide an abstract description of Mask, including its functionality and its properties. The actual implementation of Mask heavily depends on the computational model used to represent the functions. Therefore to instantiate our compiler, one needs to specify how Mask can be implemented. For the circuits model, this essentially recovers the proposal in [16] that, roughly, is based on creating a new circuit where the fixed inputs are hardwired in constant gates in such a way that they can be easily located. For the Turing machines model, we show how to implement Mask in section 6.

In what follows we provide a formal definition of Mask.

Definition 4 (Mask)**.** Mask *takes as input a tuple* (F, I, V, J) *and returns a pair* (F', I') *where:*

- F *is the description of a function with* $n \in \mathbb{N}$ *inputs.*
- $I \subseteq [n]$ *is a set of indices, of cardinality* $m \in \{0, \ldots, n\}$*, that correspond to the inputs that we want to fix.*
- $V : I \to \mathcal{M}$ *are the values of the input string that we want to fix. For example, for an input* $x \in \mathcal{M}^n$*,* V *is defined so that, for every* $i \in I$*,* $V(i) = x_i$*.*
- $J \subseteq I$ *is the subset of indices that correspond to the inputs that we want to track.*
- F' *is the description of a function with* $n' = n - m$ *inputs.*
- $J' \subseteq [|F'|]$ *is the set of indices in the description of the function* F' *that correspond to the inputs that were specified to be tracked in the set* J*. Hence the cardinality of* J' *is the same as that of* J*.*

The Mask *function should satisfy four properties:*

I. Partial application of F*. Namely, the function* F' *is the same as executing* F *partially applied on the inputs in* (I, V)*. More formally, let* $I' = [n] \setminus I = \{i'_1, ..., i'_{n'}\}$*. Then for any* $x' \in \mathcal{M}^{n'}$*, we have* $F'(x') = F(v')$ *where* $v' \in \mathcal{M}^n$ *is such that*

$$v'_i = \begin{cases} V(i) & \text{if } i \in I \\ x'_k & \text{if } i \in I' \text{ such that } i = i'_k \end{cases}$$

II. Locate the fixed inputs in F'*. The intuition behind this property is that* J *specifies a subset of indices, among the ones in the set* I*, so that for every* $j \in J$ *one can track where* $V(j)$ *is written in the description of* F'*. To this end, the* Mask *function returns another set* J' *that specifies where each of these inputs has been written. More formally, if we let* $J = \{j_1, \ldots, j_{|J|}\}$ *and* $J' = \{j'_1, \ldots, j'_{|J|}\}$*, then it holds for all* $k \in [|J|] : F'_{j'_k} = V(j_k)$*.*

III. Iterative execution of Mask*. There exists a function* Mask^* *that takes as input a tuple*

$$(F, I, V^*, J, \{G_i, V_i^*, I_i\}_{i=1}^{t-1})$$

and outputs a pair (E_t, J_t) *where*

- F *is the description of a function with* n *inputs.*
- $I \subseteq [n]$.
- $J \subseteq I$.
- $V^* : I \setminus J \to \mathcal{M}$.
- G_i *is the description of a function with* N_i *inputs.*
- $I_i \subset [N_i]$
- $V_i^* : I_i^* \to \mathcal{M}$ *for some* $I_i^* \subset I_i$.

and that satisfies the following property.

Let F *be a function with* n *inputs,* $I \subseteq [n]$*,* $J \subseteq I$ *be two subsets of indices, and* $V : I \to \mathcal{M}$ *be a function that encodes inputs to be fixed. Consider, a first execution of* Mask

$$(E_1, J_1') \leftarrow \mathsf{Mask}(F, I, V, J)$$

and then, for $i = 1$ *to* $t - 1$*, consider the iterative executions*

$$(E_{i+1}, J_{i+1}') \leftarrow \mathsf{Mask}(G_i, I_i, V_i, I_i \cap J_i')$$

where:

- *Each* G_i *is some function with* $N_i \geq |E_i|$ *inputs.*
- $I_i \subset [N_i]$ *is such that* $([N_i] \setminus I_i) \subset J_i$*. Namely, the indices of inputs that are* not *fixed (i.e. they became variable in the new function* E_{i+1}*) must be contained in* J_i'.
- $V_i : I_i \to \mathcal{M}$ *is such that for all* $j \in [|E_i|] \cap I_i$ *we have* $V_i(j) = E_{i,j}$*. This models that we are considering a partial application of* G_i *on* E_i *as first input, except that some indices are removed from its description.*

Then, it holds that

$$(E_t, J_t) = \mathsf{Mask}^*(F, I, V^*, J, \{G_i, V_i^*, I_i\}_{i=1}^{t-1})$$

with

- $V^* : I \setminus J \to \mathcal{M}$ *such that* $V^*(i) = V(i), \forall i \in I \setminus J$*. In other words,* V^* *only contains the inputs that will stay fixed until the last execution.*
- $V_i^* : I_i \setminus [|E_i|] \to \mathcal{M}$ *such that* $V_i^*(j) = V_i(j), \forall j \in I_i \setminus [|E_i|]$*. In other words,* V_i^* *only contains the additional inputs (other than* E_i*) that are provided to* G_i

In summary, the third property of the Mask *function models that, an interative application of it as above, it should be possible to create the description of the last function without knowing the original inputs with indices in the set* J.

IV. Complexity of Mask*. There are two functions* $T_{\mathsf{Mask}}(\cdot)$ *and* $S_{\mathsf{Mask}}(\cdot)$ *that determine the running time and the description size, respectively, of the function* F' *that is returned by* Mask *on input a function* F.

4.2 Our compiler

Let us now describe our compiler for turning an SKHS into an MKHS.

Definition 5 (SKHS to MKHS Compiler)**.** *Let* $\Pi_{SH} =$ (HS.Setup, HS.KeyGen, HS.Sign, HS.Eval, HS.Verif) *be an SKHS scheme for programs in a given computational model, and let* $EQ, Compose$, Mask *be three functions as in Definitions 2, 3, and 4. The compiler described below outputs an MKHS scheme* $\Pi_{MH} =$ (MH.Setup,MH.KeyGen, MH.Sign, MH.Eval, MH.Verif) *for programs in the same computational model as the SKHS one.*

- $\mathsf{pp} \leftarrow \mathsf{MH.Setup}(1^\lambda, N_t, cp)$. *The* MH.Setup *algorithm takes as input a security parameter* λ*, a bound* N_t *for the number of distinct users, and a computation cost parameter* cp*. This parameter determines some upper bounds for the acceptable computational costs of the input functions to the* MH.Eval *and* MH.Verif *algorithms in the given model of computation. It outputs* $\mathsf{pp} = \langle \mathcal{ID}, \mathsf{pp}' \rangle$ *where* $\mathcal{ID}$ *is the users identity space and* pp' *is generated by invoking the* HS.Setup *algorithm as follows.*

$$\mathsf{pp}' \leftarrow \mathsf{HS.Setup}(1^\lambda, cp')$$

where cp' *(which is derived from* cp*) determines some bounds for the computational complexity of functions supported by the* HS.Eval *and* HS.Verif *algorithms. The public parameter* pp' *includes a message space* $\mathcal{M}$*, a tag space* $\mathcal{T}$*, a signature space* $\mathcal{Y}$*, and a set of admissible functions* F *determined according to* cp'*. Note, we slightly depart from the notation of Definition 1 assuming that* MH.Setup *takes* N_t *and* cp *as additional inputs (and similarly for* HS.Setup*). This is more convenient here to show more clearly the dependence of parameters. Also it can be done without loss of generality as we could assume that there is an algorithm for each choice of these values.*

- $(sk, pk) \leftarrow \mathsf{MH.KeyGen}(\mathsf{pp})$. *The key generation takes as input the public parameter* pp*, parses it as* $\langle \mathcal{ID}, \mathsf{pp}' \rangle$*, and invokes the* $\mathsf{HS.KeyGen}(\mathsf{pp}')$ *algorithm to generate the secret/public key pair* (sk, pk)*.*

  This algorithm is run by each user with identity $id \in \mathcal{ID}$*. Throughout the paper, we use* (sk_{id}, pk_{id}) *to denote that the keys are associated to the user with identity* id*.*

- $\sigma \leftarrow \mathsf{MH.Sign}(sk, \Delta, \mathrm{m}, \ell)$. *This algorithm takes as input the secret key* sk *of the user with identity* $id \in \mathcal{ID}$*, the data set identifier* Δ*, the message* $\mathrm{m} \in \mathcal{M}$*, and the label* $\ell = (id, \tau)$ *where* $\tau \in \mathcal{T}$*, and runs the following algorithm:*

$$\sigma \leftarrow \mathsf{HS.Sign}(sk, \Delta, \mathrm{m}, \tau)$$

- $\sigma' \leftarrow \mathsf{MH.Eval}(\mathcal{P}, \Delta, \{pk_{id_i}, \sigma_i\}_{i=1}^n)$. *Let* $t \leq N_t$ *be the number of distinct public keys taken as input by the algorithm. Let* $\mathcal{P} = (F, (\ell_1, ..., \ell_n))$ *be the labeled program where* $n \geq t$ *and for each* $j \in [n]$*,* $\ell_j = (id_i, \tau_j)$ *such that* $id_i \in [t]$ *and* $\tau_j \in \mathcal{T}$*. The resulting signature is computed as follows.*

  For $t = 1$*, there is only one signing user with identity* id *and public key* pk_{id}*. Namely, all signatures* $\{\sigma_1, ..., \sigma_n\}$ *belong to this user and in the labeled program* $\mathcal{P}$*, all labels has the*

form $\ell_j = (id, \tau_j)$ with $j \in [n]$ and some $\tau_j \in \tau$. In this case, the MH.Eval *algorithm directly runs the* HS.Eval *algorithm as follows.*

$$\sigma' \leftarrow \mathsf{HS.Eval}(\mathcal{P}, \Delta, pk_{id}, \{\sigma_1, ..., \sigma_n\})$$

For $t > 1$*, there is more than one signing users with identifiers $\{id_1, ..., id_t\}$ and different public keys $\{pk_{id_1}, ..., pk_{id_t}\}$. Assume that each user with identity id_i contributes n_i messages such that $n = \sum_{i=1}^{t} n_i$. We assume (without loss of generality) that labels and their corresponding messages and signatures are ordered based on the user's identifier. Namely, the set of triples of labels, messages, and signatures belong to the user with identity id_i is $\{(\ell_j, \mathrm{m}_j, \sigma_j)\}_{j=I_s}^{I_e}$ where $I_s = 1 + \sum_{j=0}^{i-1} n_j$, $n_0 = 0$, and $I_e = \sum_{j=1}^{i} n_j$. In this case, the* MH.Eval *algorithm executes the following $t + 1$ steps.*

- ***Step 0:*** *compute $y \leftarrow F(\mathrm{m}_1, ..., \mathrm{m}_n)$ and convert F to a single output function by running $E_0 \leftarrow Compose(F, EQ^y)$. Hence, we have that*

$$E_0(m_1, ..., m_n) = \begin{cases} 1 & \text{if } F(m_1, ..., m_n) = y \\ 0 & \text{otherwise} \end{cases},$$

  E_0 works by running F on the inputs and comparing its output to the embedded value y, to check whether it is equal to y or not.

  Notice that the function E_0 takes its inputs from all users. However, recall that the HS.Eval *algorithm can be executed on functions with inputs from one user only. For this reason, in the next steps we apply repeatedly the* Mask *function to create a function E_i takes n_i inputs only from user id_i, and has inputs from users $id_{i+1}, \ldots, id_t$ fixed in its description. To do this we also exploit the property of the SKHS verification algorithm.*

- ***Step 1:*** *First, compute*

$$(E_1, J_1) \leftarrow \mathsf{Mask}(E_0, I_0, V_0, J_0)$$

  where $I_0 = \{n_1 + 1, ..., n\}$, $V_0 : I_0 \rightarrow \mathcal{M}$ such that $\forall i \in I_0 : V(i) = m_i$, and $J_0 = I_0$. Essentially, E_1 is the partial application of E_0 on the last $n - n_1$ inputs $m_{n_1+1}, \ldots, m_n$. Hence,

$$E_1(m_1, ..., m_{n_1}) = \begin{cases} 1 & \text{if } E_0(m_1, ..., m_n) = 1 \\ 0 & \text{otherwise} \end{cases},$$

  Also, by setting $J_0 = I_0$ we are asking Mask *to keep track of all the inputs from users $id_{i+1}, \ldots, id_t$, and therefore the set J_1 contains the indices in $[|E_1|]$ in which the inputs $\{m_{n_1+1}, ..., m_n\}$ have been written in the description of E_1. Now that the function E_1 takes inputs from the first user only, we execute the SKHS evaluation algorithm on it as follows.*

$$\sigma'_1 \leftarrow \mathsf{HS.Eval}\Big(\big(E_1, (\tau_1, ..., \tau_{n_1})\big), \Delta, pk_1, \{\sigma_1, ..., \sigma_{n_1}\}\Big)$$

- ***Step i,*** $2 \leq i \leq t$***:*** *From this step on, the goal is to prove that σ'_{i-1} verifies correctly with the program E_{i-1}. Let us consider the case of $i = 2$ in which one would like to verify σ'_1. The challenge is that the verifier would need to reconstruct the function E_1 whose description, however, contain inputs from all users except the first one. To eliminate*

this dependency, the following procedure is repeated for the remaining users. Let HSV_i *be the program that models the SKHS verification algorithm on the appropriate input length* $N_i = |E_{i-1}| + n_{i-1}|\tau| + |\Delta| + |pk_{i-1}| + |\sigma' + i - 1| + 1$.[1] *By the correctness of the SKHS verification we have that*

$$HSV_i\Big(\big(E_{i-1}, (\tau_s, ..., \tau_e)\big), \Delta, pk_{i-1}, \sigma'_{i-1}, 1\Big) = 1$$

where $s = (\sum_{j=1}^{i-2} n_j) + 1$ *and* $e = \sum_{j=1}^{i-1} n_j$.

Also, we denote $INP_i = \Big(\big(E_{i-1}, (\tau_s, ..., \tau_e)\big), \Delta, pk_{i-1}, \sigma'_{i-1}, 1\Big) \in \mathcal{M}^{N_i}$.

Therefore, in the i-th step we create E_i *by fixing the input tuple* INP_i *of* HSV_i except *for the values of user* id_i *that are present in the description of* E_{i-1}. *Notice that the set* J_{i-1} *records the indices where the inputs from users* $id_i, id_{i+1}, \ldots, id_t$ *are in the description of* E_i. *Let us parse this set with the following notation* $J_{i-1} = \{j_s, j_{s+1}, \ldots, j_n\}$. *The* Mask *function is executed as follows*

$$(E_i, J_i) \leftarrow \mathsf{Mask}(HSV_i, I_i, V_i, I_i \cap J_{i-1})$$

where

- $I_i \leftarrow ([|E_{i-1}|] \setminus \{j_{s+1}, \ldots, j_e\}) \cup \{|E_{i-1}| + 1, \ldots, N_i\} \subset [N_i]$.
- $V_i : I_i \rightarrow \mathcal{M}$ *such that for all* $j \in I_i$, *we have* $V_i(j) = INP_{i,j}$. *Such definition of* (I_i, V_i) *models that we are asking* Mask *to fix the whole input* INP_i *except those values that correspond to the inputs of user* id_i *that, by the correctness of the previous* Mask *execution, are in the positions* $\{j_{s+1}, \ldots, j_e\}$ *in* E_{i-1}*'s description.*

We also observe that by passing $I_i \cap J_{i-1}$ *as the last input of* Mask, *we are essentially asking* Mask *to keep track of the values in positions* $j_{e+1}, \ldots, j_n$, *and thus* J_i *contains the indices where these values have been written in the description of* E_i.

By the property I of Mask, *if we assume* $s' = s + n_{i-1}$ *and* $e' = e + n_i$, *the function* E_i *is such that*

$$E_i(m_{s'}, ..., m_{e'}) = \begin{cases} 1 \quad \textit{if} \;\; \mathsf{HS.Verif}\Big(\big(E_{i-1}(m_s, ..., m_e), (\tau_s, ..., \tau_e)\big), \\ \qquad\qquad \Delta, pk_{i-1}, \sigma'_{i-1}, 1\Big) = 1 \\ 0 \quad \textit{otherwise} \end{cases},$$

Since the function E_i *takes in only inputs of user* id_i, *we can run* HS.Eval *on it as below to create* σ'_i.

$$\sigma'_i \leftarrow \mathsf{HS.Eval}\Big(\big(E_i, (\tau_{s'}, ..., \tau_{e'})\big), \Delta, pk_i, \{\sigma_{s'}, ..., \sigma_{e'}\}\Big)$$

Finally, notice that in the last step, the function E_t *just takes its inputs from the last user and the evaluation algorithm is executed on it as follows.*

$$\sigma'_t \leftarrow \mathsf{HS.Eval}\Big(\big(E_t, (\tau_{1+\sum_{j=1}^{t-1} n_j}, ..., \tau_n)\big), \Delta, pk_t, \{\sigma_{1+\sum_{j=1}^{t-1} n_j}, ..., \sigma_n\}\Big)$$

[1] The use of an input-specific function is done for generality to capture both uniform (e.g., Turing machines) and non-uniform (e.g. circuits) computational models.

The tuple of the signatures $\hat{\sigma} = \langle \sigma'_1, ..., \sigma'_t \rangle$ is sent to the verifier as an authenticator for verifying the result y.

- $b := \mathsf{MH.Verif}\big(\mathcal{P}, \Delta, \{pk_i\}_{i=1}^{t}, \hat{\sigma}, y\big)$. *The verification algorithm takes as inputs the labeled program $\mathcal{P} = \big(F, (\ell_1, ..., \ell_n)\big)$, the public key of all users, the dataset identifier Δ, the authenticator $\hat{\sigma} = \langle \sigma'_1, ..., \sigma'_t \rangle$, and the result message $y \in \mathcal{M}$, and outputs one bit $b \in \{0, 1\}$ deciding whether the result is correct or not.*

  For $t = 1$*, the labeled program $\mathcal{P} = (F, (\ell_1, ..., \ell_n))$ takes inputs from a single user with identity id and public-key pk, and for every $j \in [n]$, $\ell_j = (id, \tau_j)$ for some $\tau_j \in \mathcal{T}$. In this case $\hat{\sigma} = \langle \sigma \rangle$ and the verifier only runs the verification algorithm in the SKHS scheme as bellow and returns b.*

$$b := \mathsf{HS.Verif}\Big(\big(F, (\tau_1, ..., \tau_n)\big), \Delta, pk, \sigma, y\Big)$$

  For $t > 1$*, the labeled program $\mathcal{P} = (F, (\ell_1, ..., \ell_n))$ takes inputs from more than one user. The verifier constructs E_0 like the Step 0 explained in the* MH.Eval *algorithm, parses $\hat{\sigma}$ to $\langle \sigma'_1, ..., \sigma'_t \rangle$, and reconstructs E_t using the public values $\langle (\tau_1, ..., \tau_n), \{pk_i\}_{i=1}^{t-1}, \Delta, \{\sigma'_i\}_{i=1}^{t-1} \rangle$, and the third property of the* Mask *function as*

$$(E_t, \phi) \leftarrow \mathsf{Mask}^*(E_0, I_0, V^*, J_0, \{HSV_i, V_i^*, I_i\}_{i=1}^{t-1})$$

  where the sets of indices I_0, J_0 are as in the MH.Eval *algorithm, V^* is the empty function, the sets I_i are defined as in* MH.Eval*, and each V_i^* encodes only the inputs of HSV_i after E_{i-1} (that are public). Finally, the verifier runs the following algorithm.*

$$b := \mathsf{HS.Verif}\big((E_t, \tau_{(\sum_{i=1}^{t-1} n_i)+1}, ..., \tau_n), \Delta, pk_t, \sigma'_t, 1\big)$$

4.3 Succinctness, correctness and security of the compiler

The succinctness, correctness, and proof of security of the compiler follows the ones given for Matrioska [16].

Succinctness Let Π_{HS} be an SKHS scheme with the succinctness l. The succinctness of the multi-key homomorphic signature scheme which is obtained by the compiler is determined by the size of σ' which is the output of the MH.Eval algorithm. As $\hat{\sigma} = (\sigma'_1, ..., \sigma'_t)$ and each σ'_i is the output of the HS.Eval algorithm, so the succinctness of the MKHS scheme is $l.t$.

Correctness Let $\Pi_{\mathsf{HS}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ is an SKHS scheme with the correctness property, then MKHS scheme $\Pi_{\mathsf{MH}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ obtained from the compiler has the correctness property according to the correctness of scheme defined in the Definition 1.

Proof. The correctness property consists of the authentication correctness and evaluation correctness. The main idea of the proof is quite similar to the proof of the Matrioska [16], with the difference that in our case the steps are generalized to any computational model and make use of the properties of the generic Mask function. Hence, the correctness of the MKHS scheme obtained from the compiler is reduced to the correctness of the SKHS scheme. The details of the proof is in Appendix A. □

Security Let Π_{HS} be a secure SKHS scheme, then the MKHS scheme Π_{MH} generated by the compiler is secure.

Now, we explain an intuition of the security proof for the MKHS generated by the compiler and details of the proof can be found in [16].

Proof. Let $\mathcal{A}$ be a PPT adversary against the Π_{MH} MKHS scheme. We show that if the adversary $\mathcal{A}$ makes a forgery in Π_{MH}, we can create another PPT adversary $\mathcal{B}$ that outputs a forgery against the SKHS scheme Π_{HS} such that $\Pr[\exp^{MKHS}_{A,\Pi_{\mathsf{MH}}}(\lambda) = 1] < t \cdot \Pr[\exp^{SKHS}_{B,\Pi_{\mathsf{HS}}}(\lambda) = 1]$ where t is the maximum number of distinct identities.

Let $\mathcal{A}$ outputs a forgery $(\mathcal{P}^*, \Delta^*, \mathrm{m}^*, \sigma'^*)$ where $\sigma'^* = (\sigma'^*_1, ..., \sigma'^*_t)$ and $\mathsf{MH.Verif}\big(\mathcal{P}^*, \Delta^*, \{pk_{id}\}_{id\in\mathcal{P}^*_{\Delta^*}}, \sigma'^*, 1\big) = 1$. For $(t = 1)$, since the MKHS scheme generated by the compiler is similar to an SKHS scheme, it is obvious that a forgery in Π_{MH} leads a forgery in Π_{HS}. For $(t > 1)$, we show how a forgery in Π_{MH} leads a forgery in Π_{HS}.

- Let $\mathcal{A}$ outputs a Type-I forgery, namely Δ^* be a new dataset identifier. In the verifier side, the $\mathsf{MH.Verif}$ algorithm invokes $\mathsf{HS.Verif}\big((E^*_t, (\tau_{n-n_t}, ..., \tau_n)), \Delta^*, \sigma'^*_t, 1\big)$ algorithm. So, if Δ^* is not queried before, then $\mathcal{B}$ outputs a Type-I forgery in Π_{HS}.

- Let $\mathcal{A}$ outputs a Type-II forgery in Π_{MH} such that $E_0(\mathrm{m}_1, ..., \mathrm{m}_n) = 0$ and $\mathsf{MH.Verif}\big((E_0, (\tau_1, ..., \tau_n)), \Delta^*, \{pk_i\}^t_{i=1}, \sigma'^*, 1\big) = 1$. This algorithm creates E_t and calls $\mathsf{HS.Verif}\big((E_t, (\tau_{n-n_t}, \ldots, \tau_n)), \Delta^*, pk_t, \sigma'^*_t, 1\big)$ algorithm which outputs 1. Therefore, in at least one step $i \in [t]$ of the compiler, there is a function E_{i-1} such that

$$\mathsf{HS.Verif}\Big(\big(E_{i-1}, (\tau_{(\sum^{i-1}_{j=1} n_j)+1}, ..., \tau_{\sum^{i}_{j=1} n_j})\big), \Delta, pk_{(\text{i-1})}, \sigma'_{(\text{i-1})}, 1\Big) = 1$$

  and $E_{i-1}(\mathrm{m}_{\sum^{i-2}_{j=1} n_j}, ..., \mathrm{m}_{\sum^{i-1}_{j=1} n_j}) = 0$. Therefore the adversary $\mathcal{B}$ can output a Type-II forgery in Π_{HS}.

- Let $\mathcal{A}$ outputs a Type-III forgery in Π_{MH} and there is at least an index $i \in [n]$ such that $\ell_i = (id_i, \tau_i) \notin \mathcal{L}_{\Delta^*}$. Since in the steps of the compiler, the functions are created using the input program and they are used in $\mathsf{HS.Verif}$ algorithm as the input function, then the Type-III forgery is unavoidable in Π_{HS}. Another case that the adversary $\mathcal{A}$ can output a type-III forgery in Π_{MH} is that there is a $\sigma^*_{i'}$ such that $\sigma^*_{i'}$ be a Type-II forgery in Π_{HS} for some identity $id_{i'}$ which is used during the verification in the compiler.

□

Remark 2. *Lai et al. [24] introduced the notion of insider-unforgeability for SKHS and MKHS and proposed constructions of these schemes from non-falsifiable assumptions. We observe that in the case our compiler is applied to an insider-unforgeable SKHS scheme, the resulting MKHS scheme would also satisfy insider unforgeability. Intuitively, this is due to the fact that an SKHS with insider unforgeability is essentially a succinct proof system. Note though that applying our compiler to an insider-unforgeable SKHS makes the purpose of the compiler less interesting. Indeed, insider unforgeable SKHS imply SNARGs, which are known to require non-falsifiable assumptions under which one could build directly a more efficient MKHS (as proposed in [24]).*

4.4 General efficiency analysis

In this section, we provide a general framework to analyze the efficiency of the MKHS scheme obtained through our compiler. In particular, we are interested in analyzing the complexity of running the MH.Eval and MH.Verif algorithms. To this end, our analysis focuses on showing the running time and the description size of the functions E_i built during the steps of the MH.Eval algorithm – this is in fact the main burden of our general compiler. This analysis clearly includes also the complexity of the last function E_t that the verifier feeds to the HS.Verif algorithm.

We stress that here we only provide a general framework to analyze these costs. As we shall see, the final complexity depends on the specific SKHS we start from, the computational model supported by it, and the Mask function.

Given a function E_i built in the i-th step of the compiler, we denote its running time with $T(E_i)$ and the size of its description with $S(E_i)$.

Let HSV be the program that realizes the HS.Verif algorithm on an input $((E, \tau_1, \ldots, \tau_n), \Delta, pk, \sigma, y)$. We assume that its running time $T(HSV)$ and description size $S(HSV)$ are determined using the following functions:

$$T(HSV) = T_V\big(S(E), T(E), n, \lambda\big), \qquad S(HSV) = S_V\big(S(E), n, \lambda\big)$$

Namely, they are both a function of HSV's input length and the security parameter, and the running time of HSV may also depend on the running time of E.
Let us now review the steps of the compiler to analyze the growth in the description size and the verification time. Let E_0 be the program we start from and let us assume it has size $S(E_0)$ and that runs in time at most $T(E_0)$ on n inputs.

- **Step 1:** We build the function E_1 using the mask function as $(E_1, J_1) \leftarrow \mathsf{Mask}(E_0, I_0, V_0, J_0)$. By the property IV of the Mask function (see Definition 4), we have

  $$T(E_1) = T_{\mathsf{Mask}}(T(E_0)), \qquad S(E_1) = S_{\mathsf{Mask}}(S(E_0))$$

- **Step i ($i \geq 2$)**: let HSV_i be the program that models the SKHS verification algorithm to be run on a labeled program of size $S(E_{i-1})$ and with running in time at most $T(E_{i-1})$ on n_{i-1} inputs. The function E_i is created using the mask function as $(E_i, J_i) \leftarrow \mathsf{Mask}(HSV_i, I_i, V_i, I_i \cap J_{i-1})$.

  By the property IV of the Mask function, we have

  $$T(E_i) = T_{\mathsf{Mask}}(T(HSV_i)), \qquad S(E_i) = S_{\mathsf{Mask}}(S(HSV_i))$$

  In turn, as mentioned earlier the time and size costs of HSV_i are

  $$\begin{aligned} T(HSV_i) &= T_V\big(S(E_{i-1}), T(E_{i-1}), n_{i-1}, \lambda\big), \\ S(HSV_i) &= S_V\big(S(E_{i-1}), n_{i-1}, \lambda\big) \end{aligned}$$

  Therefore, by combining these equations we obtain the following relation at every step $2 \leq i \leq t-1$:

  $$T(E_i) = T_{\mathsf{Mask}}(T_V\big(S(E_{i-1}), T(E_{i-1}), n_{i-1}, \lambda\big)), \tag{1}$$
  $$S(E_i) = S_{\mathsf{Mask}}(S_V\big(S(E_{i-1}), n_{i-1}, \lambda\big)) \tag{2}$$

At this point it is clear that to resolve this relation one needs to instantiate the functions T_V and S_V (that depends on the SKHS scheme used in the compiler), and the functions T_{Mask} and S_{Mask} (that depend on the Mask function, which in turn depends on the computational model).

For this reason we defer the reader to the following two sections to see how the analysis can be completed in two models of computation, that of circuits (which recovers the result of Fiore et al. [16]) and that of Turing machines (that we show how to instantiate in this work).

To ease the comparison, we assume that the function S_V can be expressed generally as below:

$$S_V(S(E), n, \lambda) = \mathfrak{a}_s(S(E), \lambda) \cdot S(E) + \mathfrak{b}_s(n, \lambda)$$

for some functions $\mathfrak{a}_s$ and $\mathfrak{b}_s$ that are both polynomials in their inputs. We can see in Section 5.1 and Section 6.4, how the computational model and implementation of the Mask function can affect the description size of the final function E_t created in the compiler. In particular, one can see that in a uniform model of computation such as Turing machines the description of the Turing machine can be independent of the input length, i.e., $S_V(S(E), n, \lambda)$ is a fixed polynomial $\mathfrak{b}_s(\lambda)$, whereas in a model like circuits the size of the circuit description is *at least* as large as the input size.

Regarding the running time of the $\mathsf{HS.Verif}$ algorithm, to ease the comparison we consider the following five cases of the function T_V:

- **Case 1**: $T_V(T, S, n, \lambda) = p(\lambda)T\log(T)$.
- **Case 2**: $T_V(T, S, n, \lambda) = p(\lambda)T$.
- **Case 3**: $T_V(T, S, n, \lambda) = cT + np(\lambda)$ for a constant $c \in \mathbb{N}$.
- **Case 4**: $T_V(T, S, n, \lambda) = S\log(T) + np(\lambda)$.
- **Case 5**: $T_V(T, S, n, \lambda) = S + np(\lambda)$.

Notably, the first case models the efficiency of the state-of-the-art SKHS scheme for circuits proposed by Gorbunov et al. [21]. In case 1, even if we assume that T_{Mask} is the identity function, it is not hard to see that the recurrence of equation (1) would resolve $T(E_t)$ having a factor λ^{t-1}, exponential in the number t of users. For this reason, we consider the additional cases 2–5 as they show the potential of our compiler and can motivate further research in designing SKHS with more efficient verification algorithms. We note that the last case 5 is the most optimistic one, as it is saying that the running time of the SKHS verification depends linearly in the description size of the function to verify, without any multiplicative factor, depending on a constant (i.e., $c \cdot S$) or the security parameter (i.e., $\lambda \cdot S$). Finally, we stress that the verification algorithm must read the function to verify, which is an input, and thus it seems unrealistic to assume a verification time smaller than the size S.

5 Our compiler in the circuits model: Matrioska

The compiler proposed by Fiore et al. [16] can be seen as an instantiation of our general compiler described in Section 4.2 when instantiated in the circuits model of computation. Below we recall the circuit model used in [16] and explain how the Mask and $\mathsf{Compose}$ functions work there.

In the circuit model, a circuit C is described using a tuple $C = (n_i, o_i, q_i, L_i, R_i, G_i)$ where

- n_i is the number of inputs,
- o_i is the number of outputs,
- q_i is the number of gates,
- L_i is a function determines the left wire for a given gate $g \in [q_i]$,
- R_i is a function determines the right wire for a given gate $g \in [q_i]$,
- G is a function that maps each gate $g \in [q_i]$ to a bit.

In Matrioska, the EQ^y, Compose, and Mask functions that are used by the compiler are modeled using circuits. The function EQ^y is modeled by a circuit that models the *xor* operation. The Compose function just binds the outputs of the first circuit to the inputs of the second circuit, and is used by the function $\mathsf{Mask}(C, I, V, J)$. In Algorithm 1 we show how the description of this provided in [16] can be seen as an instance of our Mask definition. In a nutshell, the algorithm creates a mask circuit M that outputs the inputs of the circuit C and them it runs the Compose function to binds the outputs of M to the inputs of C. We defer the reader to [16] for more details.

Algorithm 1 : Mask function in Matrioska

procedure $\mathsf{Mask}(C, I, V, J)$
  Parse C to (n, o, q, L, R, G);
  Define empty arrays $L_m[n]$, $R_m[n]$, $G_m[n]$;
  for each $i \in I$ **do**
    Set $L_m(i) = 0$, $R_m(i) = 0$, $G_m(i) = V(i)$;
  end for
  for each $i \in [n] \setminus I$ **do**
    Set $L_m(i) = 1$, $R_m(i) = 1$, $G_m(i) = 0$;
  end for
  Create a mask circuit $M = (1, n, n, L_m, R_m, G_m)$;
  Run $C' = compose(M, C)$;
  for each $i \in J$ **do**
    Set J'_i to the location of J_i in C';
  end for
  Return (C', J')
end procedure

Let us assume that (MH.Setup, MH.KeyGen, MH.Sign, MH.Eval, MH.Verif) are the algorithms generated by Matrioska. The algorithms MH.KeyGen and MH.Sign are exactly the same as the algorithms presented in Definition 5. In the inputs of the MH.Setup algorithm, the computation cost parameter cp is a pair (s, d) where s and d are upper bounds for the size and depth of the circuits supported by the generated MKHS scheme, respectively. In the MH.Eval and MH.Verif algorithms, the steps are the same as the steps described in Definition 5, but the EQ^y, Mask, Compose, and every function E_i created in the i-th step of these algorithms are modeled using a circuit.

5.1 Efficiency analysis

Fiore and Pagnin analyzed the efficiency of their compiler [16] by showing how the circuits E_i grow and what is the size of the last circuit E_t, which is the one that needs to be computed by the verifier. In their analysis, this size depends in a double exponential manner on t, roughly $S(HSV_t) \approx S(C)^{c^t}$, where c is a constant such that the SKHS verification on input a circuit C is a circuit with $p(\lambda)S(C)^c$ gates. The depth of the circuit is a function of its size. This assumption is quite general (it simply assumes that the verification circuit is a polynomial of its input size) but not very tight.

For a fair comparison with the compiler proposed in this paper, we revisit the complexity analysis of Matrioska considering the most favorable assumption regarding the complexity of the SKHS verification with respect to the input program. Even in such optimistic case, we show that the MKHS scheme obtained through the Matrioska compiler implies a blowup that is *at least* exponential in the number t of users.

In the circuit model we take the circuit size (i.e., the number of gates) as a measure for the running time, while the description size of a circuit with q gates is assumed to be $\approx q \log q$, following the model adopted in [16]. Therefore, applying the case 5, in what follows we assume that

$$T(HSV_i) = T_V(T(E_{i-1}), S(E_{i-1}), n_{i-1}, \lambda) = S(E_{i-1}) + n_{i-1} \cdot p(\lambda)$$

and $S(HSV_i) = T(HSV_i)\log(T(HSV_i))$.

Let us now review how the compiler proceeds.

- **Step 1**: In the first step, the circuit E_1 is created by running the Mask function that invokes $\mathsf{Compose}(M_1, E_0)$. The size of this circuit is $S(E_1) = S(E_0) + n$ where n is the total number of inputs of E_1 and the number of gates in the masking circuit M_1.

- **Step i** ($2 \leq i \leq t-1$): Let HSV_i be the description of the circuit that models the verification algorithm of the SKHS scheme that is created in the i-th step of the compiler. This circuit is to be run on the input

$$INP_i = \Big(\big(E_{i-1}, (\tau_{1+(i-1)n'}, ..., \tau_{in'})\big), \Delta, pk_{i-1}, \sigma'_{i-1}, 1\Big)$$

  Next, E_i is built using the Mask function that internally computes $E_i = \mathsf{Compose}(M_i, HSV_i)$ where M_i is a "mask" circuit that is as large as the input INP_i above (see Algorithm 5). Let $l(\lambda)$ be a $poly(\lambda)$ such that $l(\lambda) = |\Delta| + |pk| + |\sigma'| + 1$. The size of the circuit E_i is

$$\begin{aligned} S(E_i) &= S(M_i) + S(HSV_i) \\ &= S(E_{i-1}) + n_{i-1}|\tau| + l(\lambda) + T(HSV_i)\log(T(HSV_i)) \\ &= S(E_{i-1}) + n_{i-1}|\tau| + l(\lambda) \\ &\quad + (S(E_{i-1}) + n_{i-1} \cdot p(\lambda))\log(S(E_{i-1}) + n_{i-1} \cdot p(\lambda)) \end{aligned}$$

  As one can see, from the above equation we can easily derive a lower bound

$$S(E_i) \geq 2 \cdot S(E_{i-1})$$

  and with an easy inductive proof we also get that for every $1 \leq i \leq t$

$$S(E_i) \geq 2^{i-1} \cdot S(E_0)$$

Hence, we obtain that in the Matrioska compiler the complexity growth in case 5 is still exponential in the number of users, namely

$$\begin{aligned} S(E_t) &\geq 2^{t-1} \cdot S(E_0) \\ T(E_t) &\geq S(E_{t-1}) + n_{t-1}p(\lambda) \geq 2^{t-2} \cdot S(E_0) + n_{t-1}p(\lambda) \end{aligned}$$

6 Instantiating our compiler for Turing machines

In this section, we show how to instantiate our compiler of Section 4.2 for SKHS that support programs modeled as Turing Machines (TM). To this end, we show how the building blocks of section 4.1 can be instantiated in the TM model.

In particular, a crucial difference with the Matrioska compiler of [16] is a more efficient implementation of the Mask function that, *independently of the complexity of the SKHS verification algorithm*, avoids an exponential blowup in the size of the TMs E_i built during the steps of the compiler.

In Matrioska, the exponential blow up in the circuit size is mainly due to "gluing" the Mask circuit to the description of the circuit HSV_i used in each step. The size of the Mask circuit depends on the input size of HSV_i, that in turn includes the size of the circuit E_{i-1} created in the previous step.

In the case of the Turing machines instantiation, we overcome this blowup by defining the Turing machines in such a way that the input values are located at the beginning of the input tape. This way we can finding and remove the input values in each step in a much simpler way than in Matrioska as we can simply "cut" the tape from the left at an appropriate location. In the next section we begin by describing our TM model.

6.1 Turing machines conventions

We model the functions in our compiler by using multi-tape Turing machines. Since every multi-tape Turing machine has an equivalent single-tape Turing machine [27], all of the multi-tape Turing machines used in this paper can be realized by transforming them to a traditional single-tape Turing machine. A multi-tape Turing machine is a 7-tuple $\langle \mathrm{Q}, \Sigma, \Gamma, \delta, \mathrm{q_s}, \mathrm{q_a}, \mathrm{q_r} \rangle$ with the following semantics:

- Q is the set of all states.
- Σ is the input alphabet (the union of all input alphabets for all tapes i.e. $\Gamma = \bigcup_{i=1}^{k} \Gamma_i$).
- $\Gamma = \Sigma \cup \{\mathrm{B}, \diamondsuit, \rhd, \lhd\}$ is the tape alphabet. It is the union of the input alphabet and four special symbols, which are not in the input alphabet. B symbol is used for determining the blank cells, $\diamondsuit$ symbol is used for separating values on the tape, $\rhd$ symbol is used for determining the beginning of the values, and $\lhd$ symbol is used for determining the end of the values on a tape.
- $\mathrm{q_s} \in \mathrm{Q}$ is the start state.
- $\mathrm{q_a} \subseteq \mathrm{Q}$ is the set of accept states.

- $q_r \subseteq Q$ is the set of reject states.
- $\delta : Q \times (\Gamma)^k \rightarrow Q \times (\Gamma)^k \times \{\mathcal{R}, \mathcal{L}, \mathcal{S}\}^k$ is the transition function where k is the number of tapes used in a Turing machine and $\mathcal{R}$, $\mathcal{L}$, and $\mathcal{S}$ stand for Right, Left, and Stop, respectively.

We use TM instead of Turing machine for simplicity throughout the paper. For describing a TM, we add a new part to the general description of the multi-tape TMs. The description of TMs used in our compiler contains

$$TM^{(i)} = \langle cv, Q^{(i)}, \Sigma^{(i)}, \Gamma^{(i)}, \delta^{(i)}, q_s^{(i)}, q_a^{(i)}, q_r^{(i)} \rangle$$

where i is a TM identifier and cv is the constant inputs of a TM. The transition function of TMs used in our compiler is $\delta^{(i)} : Q^{(i)} \times (\Gamma^{(i)})^4 \rightarrow Q^{(i)} \times (\Gamma^{(i)})^4 \times \{\mathcal{L}, \mathcal{R}, \mathcal{S}\}^4$ and all TMs have four tapes as:

- **Variables tape**: This tape contains variable inputs of TMs.
- **Constants tape**: This tape is read-only and contains constant inputs of TMs and is determined in the description of TMs. Intuitively, this tape contains values that are hardwired in the TM description.
- **Work tape**: This tape is used during the execution of TMs.
- **Outputs tape**: This tape contains the outputs that are determined after the execution of a TM.

Each of these four tapes has a head that can read/write one symbol at a time. The beginning (resp. end) of the values on the tapes is determined by the tape symbol $\triangleright$ (resp. $\triangleleft$).

Without loss of generality, in our compiler we assume that all the TMs have the following behavior. Before executing a computation, the TM copies the content of the variable tape to the work tape, then appends the values of the constant tape on the work tape, and terminates the work tape with the $\triangleleft$ symbol. Then, the TM executes its transition function on the work tape without needing to read the variable and constant tapes. The output of the TM is the content of the output tape when the TM halts. At the start state of TMs, we assume the heads should be at the beginning of the values on the tapes. In halting states, the head of the output tape comes back to the beginning of the output values, and the values on the work tape are erased (because, in the sequential composition of TMs, the next TM works on the same work tape).

6.2 TM-based instantiation of our compiler's building blocks

In the instantiation of the compiler proposed in Definition 5 for Turing machines model of computation, we use three families of four-tape Turing machines: (1) a family of Turing machines for modeling the main functions to be evaluated, (2) a family of Turing machines for modeling the equality check operation, and (3) a family of Turing machines for modeling the verification algorithm of a single-key homomorphic signature. We give more details on these machines below:

- $TM^{(p)}$: This machine runs the program $p(x_1, ..., x_n)$ and writes its output on the output tape. The input values $\{x_1, ..., x_n\}$ can be constant or variable. The constant inputs are specified in $cv^{(p)}$. In a situation where all inputs are variable, this tape can be empty. The description of this machine is

$$\mathrm{TM}^{(\mathrm{p})} = \langle \mathrm{cv}^{(\mathrm{p})}, \mathrm{Q}^{(\mathrm{p})}, \Sigma^{(\mathrm{p})}, \Gamma^{(\mathrm{p})}, \delta^{(\mathrm{p})}, {\mathrm{q_s}}^{(\mathrm{p})}, {\mathrm{q_a}}^{(\mathrm{p})}, {\mathrm{q_r}}^{(\mathrm{p})} \rangle.$$

- $\mathrm{TM}^{(\mathrm{EQ}^y)}$: This machine simply checks if a variable input on the variable tape is equal to the constant input y (it is in $\mathrm{cv}^{(\mathrm{EQ}^y)}$) or not. If the equality check is satisfied, it writes the value 1 otherwise writes the value 0 on the output tape. The description of this TM is

$$\mathrm{TM}^{(\mathrm{EQ}^y)} = \langle \mathrm{cv}^{(\mathrm{EQ}^y)}, \mathrm{Q}^{(\mathrm{EQ}^y)}, \Sigma^{(\mathrm{EQ}^y)}, \Gamma^{(\mathrm{EQ}^y)}, \delta^{(\mathrm{EQ}^y)}, {\mathrm{q_s}}^{(\mathrm{EQ}^y)}, {\mathrm{q_a}}^{(\mathrm{EQ}^y)}, {\mathrm{q_r}}^{(\mathrm{EQ}^y)} \rangle.$$

- $\mathrm{TM}^{(\mathrm{v})}$: This Turing machine models the verification algorithm of a sufficiently expressive single-key homomorphic signature. It takes as constant inputs (they are in $\mathrm{cv}^{(\mathrm{v})}$), the description of a Turing machine $\mathrm{TM}^{(\mathrm{i})}$, the public key of the verifier pk, the dataset identifier Δ, the result bit b, and the signature of the result σ. If the verification is satisfied, the value 1 otherwise the value 0 is written on the output tape. The description of this machine is

$$\mathrm{TM}^{(\mathrm{v})} = \langle \mathrm{cv}^{(\mathrm{v})}, \mathrm{Q}^{(\mathrm{v})}, \Sigma^{(\mathrm{v})}, \Gamma^{(\mathrm{v})}, \delta^{(\mathrm{v})}, {\mathrm{q_s}}^{(\mathrm{v})}, {\mathrm{q_a}}^{(\mathrm{v})}, {\mathrm{q_r}}^{(\mathrm{v})} \rangle.$$

We also define two deterministic functions Compose and Mask that operate on Turing machines and create a new Turing machine as their output.

6.2.1 TM-based composition function

A composition function, Compose, is a function that takes as input two TMs and combines them to make a new TM. Let

$$\mathrm{TM}^{(1)} = \langle \mathrm{cv}^{(1)}, \mathrm{Q}^{(1)}, \Sigma^{(1)}, \Gamma^{(1)}, \delta^{(1)}, {\mathrm{q_s}}^{(1)}, {\mathrm{q_a}}^{(1)}, {\mathrm{q_r}}^{(1)} \rangle,$$

and

$$\mathrm{TM}^{(2)} = \langle \mathrm{cv}^{(2)}, \mathrm{Q}^{(2)}, \Sigma^{(2)}, \Gamma^{(2)}, \delta^{(2)}, {\mathrm{q_s}}^{(2)}, {\mathrm{q_a}}^{(2)}, {\mathrm{q_r}}^{(2)} \rangle,$$

are two TMs and $\mathrm{Q}^{(1)} \cap \mathrm{Q}^{(2)} = \emptyset$. To construct $\mathrm{TM}^{(3)} = \langle \mathrm{cv}^{(3)}, \mathrm{Q}^{(3)}, \Sigma^{(3)}, \Gamma^{(3)}, \delta^{(3)}, {\mathrm{q_s}}^{(3)}, {\mathrm{q_a}}^{(3)}, {\mathrm{q_r}}^{(3)} \rangle$, the function Compose takes $\mathrm{TM}^{(1)}$ and $\mathrm{TM}^{(2)}$ as the inputs and specifies the following parameters:

- $\mathrm{cv}^{(3)} = \mathrm{cv}^{(1)} \mid\mid \mathrm{cv}^{(2)}$,
- $\mathrm{Q}^{(3)} = \mathrm{Q}^{(1)} \cup \mathrm{Q}' \cup \mathrm{Q}^{(2)}$,
- $\Sigma^{(3)} = \Sigma^{(1)} \cup \Sigma^{(2)}$,
- $\Gamma^{(3)} = \Gamma^{(1)} \cup \Gamma^{(2)}$,
- $\delta^{(3)} = \delta^{(1)} \cup \delta' \cup \delta^{(2)}$,
- ${\mathrm{q_s}}^{(3)} = {\mathrm{q_s}}^{(1)}$,
- ${\mathrm{q_a}}^{(3)} = {\mathrm{q_a}}^{(2)}$,
- ${\mathrm{q_r}}^{(3)} = {\mathrm{q_r}}^{(2)}$.

To complete the construction, we need to determine the parameter δ' and Q' (the rest of the

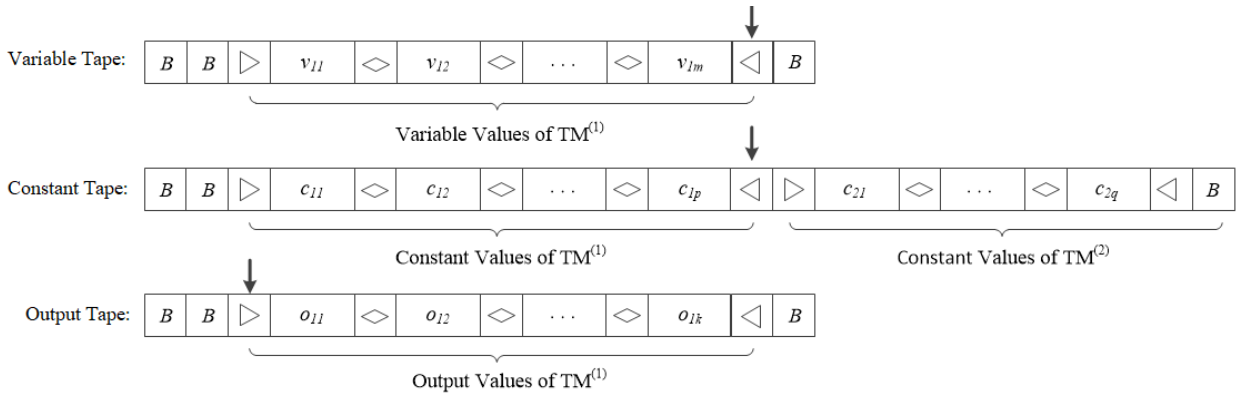

Figure 1: Values of the tapes after the execution of $\mathrm{TM}^{(1)}$.

parameters are available). The transition function δ' contains essential transitions for going from the halting states of $\mathrm{TM}^{(1)}$ to the start state of $\mathrm{TM}^{(2)}$. The contents of the tapes after the execution of $\mathrm{TM}^{(1)}$, are shown in Figure 1.

At first, δ' should copy the content of the output tape (the output of $\mathrm{TM}^{(1)}$) to the end of the variable tape. It should move the head of the output tape to the right, copy the output values to the variable tape, remove the output values from the output tape, and finally return the head of the variable tape to the beginning of the values belongs to $\mathrm{TM}^{(2)}$. The transition function δ' for 4 tapes is defined as follows,

$$\delta^{'}:\left(\{\mathrm{q_a}^{(1)},\mathrm{q_r}^{(1)}\}\cup \mathrm{Q}^{'}\right)\times\left(\Gamma^{(3)}\right)^4\mapsto\left(\{\mathrm{q_s}^{(2)}\}\cup \mathrm{Q}^{'}\right)\times\left(\Gamma^{(3)}\right)^4\times\{\mathcal{L},\mathcal{R},\mathcal{S}\}^4,$$

where $\mathrm{Q}^{'}=\{q_1',q_2'\}$ and for each $(v,c,w,o)\in\left(\Sigma^{(3)}\right)^4$. The set of transitions in δ' are as follows:

- At the end of the execution of $\mathrm{TM}^{(1)}$, the work tape is empty and the location of the heads are shown in Figure 1. This transition moves the head of the variable and constant tapes to the right.

  $\left(q,(\triangleleft,\triangleleft,\mathrm{B},\triangleright)\right)\mapsto\left(q,(\triangleleft,\triangleleft,\mathrm{B},\triangleright),(\mathcal{R},\mathcal{R},\mathcal{S},\mathcal{S})\right)\quad where\quad q\in\{\mathrm{q_r}^{(1)}\cup\mathrm{q_a}^{(1)}\}$

- This transition writes $\triangleright$ symbol on the variable tape for determining the beginning of the variable inputs of $\mathrm{TM}^{(2)}$ and moves the head of variable tape and output tape to the right.

  $\left(q,(\mathrm{B},\triangleright,\mathrm{B},\triangleright)\right)\mapsto\left(q_1',(\triangleright,\triangleright,\mathrm{B},\mathrm{B}),(\mathcal{R},\mathcal{S},\mathcal{S},\mathcal{R})\right)\quad where\quad q\in\{\mathrm{q_r}^{(1)}\cup\mathrm{q_a}^{(1)}\}$

- This transition copies the content of the output tape to the variable tape and writes the blank symbol on the output tape.

  $\left(q_1',(\mathrm{B},\triangleright,\mathrm{B},o)\right)\mapsto\left(q_1',(o,\triangleright,\mathrm{B},\mathrm{B}),(\mathcal{R},\mathcal{S},\mathcal{S},\mathcal{R})\right)$

- When the output tape reaches the end of the output values determined with $\triangleleft$, the head of the variable tape returns to the beginning of the variable inputs of $\mathrm{TM}^{(2)}$.

  $\left(q_1',(\mathrm{B},\triangleright,\mathrm{B},\triangleleft)\right)\mapsto\left(q_2',(\triangleleft,\triangleright,\mathrm{B},\mathrm{B}),(\mathcal{L},\mathcal{S},\mathcal{S},\mathcal{S})\right)$
  $\left(q_2',(v,\triangleright,\mathrm{B},\mathrm{B})\right)\mapsto\left(q_2',(v,\triangleright,\mathrm{B},\mathrm{B}),(\mathcal{L},\mathcal{S},\mathcal{S},\mathcal{S})\right)$

- when the head of the variable tape reaches the beginning of the variable inputs of $\mathrm{TM}^{(2)}$, this transition moves to the start state of $\mathrm{TM}^{(2)}$.

  $(q_2', (\triangleright, \triangleright, \mathrm{B}, \mathrm{B})) \mapsto ({\mathrm{q_s}}^{(2)}, (\triangleright, \triangleright, \mathrm{B}, \mathrm{B}), (\mathcal{S}, \mathcal{S}, \mathcal{S}, \mathcal{S}))$

6.2.2 TM-based Mask function

Here we show a realization of the Mask function for our TM model. For simplicity, our realization works for a slightly more restricted case than that of Definition 4: we assume that (I, V) always encodes a *suffix* of the input, rather than an arbitrary subset. Note that this is sufficient when instantiating our compiler of section 4.2 in our TM model as the constant tape is always at the beginning of the description and then, by the convention mentioned in section 6.2, the TM works on the variable input followed by the constant input.

The function $\mathsf{Mask}(\mathrm{TM}^{(1)}, I_1, V_1, J_1)$ takes as input the description of a Turing machine $\mathrm{TM}^{(1)} = \langle \mathrm{cv}^{(1)}, \mathrm{Q}^{(1)}, \Sigma^{(1)}, \Gamma^{(1)}, \delta^{(1)}, {\mathrm{q_s}}^{(1)}, {\mathrm{q_a}}^{(1)}, {\mathrm{q_r}}^{(1)} \rangle$ that accepts inputs of length $n^{(1)}$, a set $I_1 = \{s+1, \ldots, n^{(1)}\}$ for some $s \geq 0$, a function $V_1 : I_1 \to \mathcal{M}$ and a set $J_1 \subset I_1$. Notice that the integer s essentially denotes where to "cut" the input so that only inputs after s are fixed. Mask outputs a new Turing machine $\mathrm{TM}^{(2)} = \langle \mathrm{cv}^{(2)}, \mathrm{Q}^{(2)}, \Sigma^{(2)}, \Gamma^{(2)}, \delta^{(2)}, {\mathrm{q_s}}^{(2)}, {\mathrm{q_a}}^{(2)}, {\mathrm{q_r}}^{(2)} \rangle$ and a set J_2 that are built as follows. The constant tape $\mathrm{cv}^{(2)}$ is $\mathrm{cv}^{(2)} = \big(V_1(s+1), \ldots, V_1(n^{(1)}), \mathrm{cv}^{(1)}\big)$, while the rest of the TM stays the same, i.e., $\mathrm{Q}^{(2)} = \mathrm{Q}^{(1)}$, $\Sigma^{(2)} = \Sigma^{(1)}$, $\Gamma^{(2)} = \Gamma^{(1)}$, $\delta^{(2)} = \delta^{(1)}$, ${\mathrm{q_s}}^{(2)} = {\mathrm{q_s}}^{(1)}$, ${\mathrm{q_a}}^{(2)} = {\mathrm{q_a}}^{(1)}$, ${\mathrm{q_r}}^{(2)} = {\mathrm{q_r}}^{(1)}$. The set J_2 is defined as $\{j - s : j \in I_1\}$. Namely, we are simply shifting the indices in I_1, as in $\mathrm{cv}^{(2)}$ those values are at the beginning.

In other words, we are placing at the beginning of the constant tape of $\mathrm{TM}^{(2)}$ the suffix of the input encoded by (I_1, V_1), let us call it v, that we want to fix. Since our TMs always copy the variable inputs followed by the constant tape in the work tape, we obtain that executing $\mathrm{TM}^{(2)}$ on an s-long input v' works the same as executing $\mathrm{TM}^{(1)}$ on (v', v).

We define the following theorem to show that our TM-based Mask function satisfies the four properties of Mask function defined in Definition 4 (when executed on suffixes of inputs).

Theorem. *The TM-based* Mask *function satisfies the properties of the general* Mask *function defined in Definition 4 when executed on sets I_1 of the form $I_1 = \{s+1, \ldots, n^{(1)}\}$ for some $s \geq 0$.*

Let $(\mathrm{TM}^{(2)}, J_2) = \mathsf{Mask}(\mathrm{TM}^{(1)}, I_1, V_1, J_1)$ where $\mathrm{TM}^{(1)}$ and $\mathrm{TM}^{(2)}$ have $n^{(1)}$ and $n^{(2)} \leq n^{(1)}$ inputs, respectively. We recall each property defined in Definition 4 and prove that the TM-based Mask function satisfies each of them.

I. Partial application of $\mathrm{TM}^{(1)}$. Let $I_2 = [n^{(1)}] \setminus I_1 = \{i_{2,1}, ..., i_{2,n^{(2)}}\}$. Then for any $x' \in \mathcal{M}^{n^{(2)}}$, we have $\mathrm{TM}^{(2)}(x') = \mathrm{TM}^{(1)}(v')$ where $v' \in \mathcal{M}^{n^{(1)}}$ is such that

$$v'_i = \begin{cases} V_1(i) & \text{if } i \in I_1 \\ x'_k & \text{if } i \in I_2 \text{ such that } i = i_{2,k} \end{cases}$$

Proof. First, notice that by construction $I_2 = \{1, \ldots, s\}$ and $n^{(2)} = s$. Next, according to the Turing machine conventions explained in Section 6.1, before executing $\mathrm{TM}^{(2)}$, the content of its variable tape and constant tape are written on its working tape, receptively. Thus, the content of the working tape of $\mathrm{TM}^{(2)}$ will be $\big(x'_1, ..., x'_s, V_1(s+1), ..., V_1(n^{(1)}), CV_1\big)$. According to the definition of TM-based Mask function, we have $\mathrm{Q}^{(2)} = \mathrm{Q}^{(1)}$, $\Sigma^{(2)} = \Sigma^{(1)}$, $\Gamma^{(2)} = \Gamma^{(1)}$,

$\delta^{(2)} = \delta^{(1)}$, ${q_s}^{(2)} = {q_s}^{(1)}$, ${q_a}^{(2)} = {q_a}^{(1)}$, ${q_r}^{(2)} = {q_r}^{(1)}$. Thus, executing $TM^{(2)}$ on x' is the same as executing $TM^{(1)}$ on the input $v' = \big(x'_1, ..., x'_s, V_1(s+1), ..., V_1(n^{(1)})\big)$ (since it is also written on the working tape and $cv^{(1)}$ is appended to it). □

II. Locate the fixed inputs in $TM^{(2)}$.

Proof. In the Turing model used in this paper, the constant inputs of a Turing machine are located in the beginning of its description. Thus, finding the new indices of the fixed inputs in $TM^{(2)}$ is a simple work which is done by shifting the indices in J_1 such that $J_2 = \{j - s : j \in I_1\}$ where s is where the inputs are cut from the constant tape. □

III. Iterative execution of Mask. This property of the Mask function models that, by iterative execution of Mask function as explained in the third property of Definition 4, there is a TM-base function Mask^* that can create the description of the last Turing machine without knowing the original inputs with indices in the set J.

Proof. Let $(TM_t, J_t) = \mathsf{Mask}^*(TM, I, cv^*, J, \{G_i, cv_i^*, I_i\}_{i=1}^{t-1})$ where

- $TM = \langle cv, Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r\rangle$ is a Turing machine with n inputs.
- $I \subseteq [n]$
- $J \subseteq I$
- cv^* contains the inputs of TM that will stay fixed until the last execution.
- Each $G_i = \langle {cv}^{(i)}, {Q}^{(i)}, {\Sigma}^{(i)}, {\Gamma}^{(i)}, {\delta}^{(i)}, {q_s}^{(i)}, {q_a}^{(i)}, {q_r}^{(i)}\rangle$ is some Turing machine with $N_i \geq |TM_i|$ inputs.
- $I_i \subset [N_i]$ is such that $([N_i] \setminus I_i) \subset J_i$.
- cv_i^* is the set of fixed values that only contains the additional inputs (other than TM_i) that are provided to G_i

In the iterative execution of the TM-based Mask function, the description of TM_t will be as follows.

$$\begin{aligned} TM_t \;=\; & \big\langle ..., \big\langle \big(\\ & \big\langle \big(cv^*\big), Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r\big\rangle, \\ & , cv_1^*\big), {Q}^{(1)}, {\Sigma}^{(1)}, {\Gamma}^{(1)}, {\delta}^{(1)}, {q_s}^{(1)}, {q_a}^{(1)}, {q_r}^{(1)}\big\rangle, ...\big\rangle \\ & , cv_t^*), {Q}^{(t)}, {\Sigma}^{(t)}, {\Gamma}^{(t)}, {\delta}^{(t)}, {q_s}^{(t)}, {q_a}^{(t)}, {q_r}^{(t)}\big\rangle \end{aligned}$$

There is a sample pseudocode in Algorithm 2 that can create this description without knowing the original inputs.

Algorithm 2 : Mask^* function in TM-based compiler

procedure $\mathsf{Mask}^*(\mathrm{TM}, I, \mathrm{cv}^*, J, \{G_i, {\mathrm{cv}_i}^*, I_i\}_{i=1}^{t-1})$
  parse TM to $\langle \mathrm{cv}^*, \mathrm{Q}, \Sigma, \Gamma, \delta, \mathrm{q_s}, \mathrm{q_a}, \mathrm{q_r}\rangle$;
  set $header =$ "";
  set $body =$ "cv*), $\mathrm{Q}, \Sigma, \Gamma, \delta, \mathrm{q_s}, \mathrm{q_a}, \mathrm{q_r}\rangle$";
  for $i = 1; i < t$ **do**
    $header = header$ || "$\langle($";
    parse G_i to $\langle {\mathrm{cv}_i}^*, \mathrm{Q}^{(i)}, \Sigma^{(i)}, \Gamma^{(i)}, \delta^{(i)}, {\mathrm{q_s}}^{(i)}, {\mathrm{q_a}}^{(i)}, {\mathrm{q_r}}^{(i)}\rangle$;
    $body = body$ || ", ${\mathrm{cv}_i}^*$), $\mathrm{Q}^{(i)}, \Sigma^{(i)}, \Gamma^{(i)}, \delta^{(i)}, {\mathrm{q_s}}^{(i)}, {\mathrm{q_a}}^{(i)}, {\mathrm{q_r}}^{(i)}\rangle$";
  end for
  $\mathrm{TM}^{(t)} = header || body$;
  Set $s = I_{(t-1,1)} - 1$;
  Set $J_t = \{j - s : j \in I_{t-1}\}$;
  return (TM^t, J_t)
end procedure

□

IV. Complexity of Mask. There are two functions $T_{\mathsf{Mask}}(\cdot)$ and $S_{\mathsf{Mask}}(\cdot)$ that determine the running time and the description size, respectively, of the Turing machine $\mathrm{TM}^{(2)}$ that is returned by Mask on input a function $\mathrm{TM}^{(1)}$.

Proof. At run time, the variable and constant inputs are written on the work tape and the Turing machine executes on the work tape. Since the sum of the constant and variable inputs size is the same in $\mathrm{TM}^{(1)}$ and $\mathrm{TM}^{(2)}$ and $\delta^{(1)} = \delta^{(2)}$, we have $T_{\mathsf{Mask}}(\mathrm{TM}^{(2)}) = T(\mathrm{TM}^{(1)})$. Regarding to the description size, the TM-based Mask adds $n^{(1)} - s$ value from the constant tape and $S_{\mathsf{Mask}}(\mathrm{TM}^{(2)}) = S(\mathrm{TM}^{(1)}) + n^{(1)} - s$. □

6.3 Complexity analysis of the TM-based compiler

In this section, we describe the effect of using the TM computational model on the complexity of the proposed compiler compared to using the circuit model of computation. We define some notations used for complexity analysis in Table 1. For the TM instantiation we call $\mathrm{TM}^{(i)}$ the TM instantiation of the function E_i generated in the i-th step of the compiler.

Table 1: Notation used in the complexity analysis in Turing based compiler

Notation	Description
S_p	the size of the program $\mathrm{TM}^{(\mathrm{p})}$ to be verified.
T_p	the running time of $\mathrm{TM}^{(\mathrm{p})}$.
S_{EQ^y}	the size of $\mathrm{TM}^{(\mathrm{EQ}^{\mathrm{y}})}$.
$\mathfrak{b}_s(\lambda)$	the size of $\mathrm{TM}^{(\mathrm{v})}$.
S_{cmp}	the additional three states and six transitions in the $\mathsf{Compose}$ function.
S_i	the size of $\mathrm{TM}^{(i)}$ created in the i-th step of the compiler.
T_i	the running time of $\mathrm{TM}^{(i)}$ created in the i-th step of the compiler.
TV_i	the verification time of $\mathrm{TM}^{(i-1)}$.

6.4 Growth in the description size

Let $l(\lambda)$ be a polynomial function expressing the size of a tuple $\langle \Delta, pk, \sigma', 1 \rangle$, and let the size of the TM expressing the verification algorithm of the SKHS be a fixed polynomial, i.e. $\mathfrak{b}_s(\lambda)$. In what follows we analyze the growth of the TMs used through the steps of our compiler. In our compiler $\mathrm{TM}^{(0)}$ is created using the compose function and its size is $S_0 = S_p + S_{EQ^y} + S_{cmp} + |y|$. At step 1, the machine $\mathrm{TM}^{(1)}$ obtained from the application of Mask is essentially the same as $\mathrm{TM}^{(0)}$ with the inputs of all the users but the first one in the constant tape. Hence, its size is

$$S_1 = S_0 + n - n_1.$$

At step i, with $i \geq 2$, the compiler uses a machine $\mathrm{TM}^{(i)}$ whose size is that of $\mathrm{TM}^{(\mathrm{v})}$, i.e., $\mathfrak{b}_s(\lambda)$, plus what is written in the constant tape, which is: the description of $\mathrm{TM}^{(i-1)}$ with the first n_i inputs removed, the n_{i-1} labels and the tuple consisting of public key, signature and output bit, which has size $l(\lambda)$.

$$S_i = \text{Input size} + \text{program size} \quad \text{(in step i)}$$

More formally, we claim that

$$S_i = \begin{cases} S_0 + n - n_1 & i = 1 \\ S_0 + n - n_i + (|\tau| - 1)\sum_{j=1}^{i-1} n_j + (i-1)(\mathfrak{b}_s(\lambda) + l(\lambda)) & 1 \leq i \leq t \end{cases},$$

We prove this claim as follows.

- **Step 1**: This follows by the construction of $\mathrm{TM}^{(1)}$ as already mentioned above.
- **Step** $i \geq 2$: we show the claim by induction. Let us assume that

$$S_{i-1} = S_0 + n - n_{i-1} + (|\tau| - 1)\sum_{j=1}^{i-2} n_j + (i-2)(\mathfrak{b}_s(\lambda) + l(\lambda))$$

  In every step i for $i \geq 2$, the machine $\mathrm{TM}^{(i)}$ is created by adding to the constant tape of $\mathrm{TM}^{(\mathrm{v})}$ the description of $\mathrm{TM}^{(i-1)}$ without the n_i inputs of user i that in its constant tape $\mathrm{cv}^{(i-1)}$. Hence we have

$$\begin{aligned} S_i &= \mathfrak{b}_s(\lambda) + S_{i-1} + n_{i-1}|\tau| + l(\lambda) - n_i \\ &= S_0 + n - n_{i-1} + (|\tau| - 1)\sum_{j=1}^{i-2} n_j + (i-2)(\mathfrak{b}_s(\lambda) + l(\lambda)) + \\ &\quad \mathfrak{b}_s(\lambda) + n_{i-1}|\tau| + l(\lambda) - n_i \\ &= S_0 + n - n_i + (|\tau| - 1)\sum_{j=1}^{i-1} n_j + (i-1)(\mathfrak{b}_s(\lambda) + l(\lambda)) \end{aligned}$$

  which equals the value to be proven.

We notice that for $i = t$, S_t can be simplified with

$$S_t = S_0 + |\tau|(n - n_t) + (t-1)(\mathfrak{b}_s(\lambda) + l(\lambda))$$

This bound establishes our claim that *the size of the last Turing machine depends only linearly on the number of the users.*

6.5 Running time analysis

For analyzing the running time in our compiler, we explain the growth in the running time for different cases explained in Section 4.4. First of all, we analyzed that the Case 1 where $TV_i = \mathfrak{a}(\lambda)T_{i-1}\log(T_{i-1})$ corresponds to the way our compiler can be instantiated using the existing SKHS construction of Gorbunov et al. [21]. Then, in Section 6.5.2, we generally analyze the verification cost in other cases.

6.5.1 Efficiency of constructing the MKHS from the existing SKHS

The SKHS scheme proposed by Gorbunov et al. [21] can evaluate any circuit with bounded depth. The main cost of the verification algorithm in this scheme is the homomorphic evaluation of the circuit to be verified. This is done gate-by-gate, and each gate evaluation has a fixed $p(\lambda) = \mathsf{poly}(\lambda)$ cost. Therefore, for a circuit with N gates, the verification cost is $N \cdot p(\lambda)$. This scheme can be used for programs represented as Turing machines by using the following result, which shows how to represent a Turing machine using a Boolean circuit.

Theorem 1. *Fischer and Pippenger [17] have shown that a $T(n)$ time-bounded Turing machine (TM) can be simulated on n bits by a combinational (Boolean) circuit with $O(T(n)\cdot\log T(n))$ gates.*

Thinking of the verification algorithm of [21] as a Turing machine that evaluates a circuit in $\mathsf{poly}(\lambda) \cdot N$ steps, and by using the above theorem we get that $TV = p(\lambda)T_p\log(T_p)$ for a fixed polynomial $p(\lambda) = \mathsf{poly}(\lambda)$, where TV and T_p are the running time of the verification algorithm and input program, respectively. In this case we claim that the verification time in the i-th step is

$$TV_i \leq \begin{cases} p(\lambda)\cdot T_p\log(T_p) & i = 1 \\ T_p\cdot 5^{i-1}(i-1)!\cdot(p(\lambda)\log(T_p))^i & 2 \leq i \leq t \end{cases}$$

Proof. The case for $i = 1$ follows immediately by construction. For $2 \leq i \leq t$ we prove the claim by induction.

- For $i = 2$, the verification time of $\mathrm{TM}^{(2)}$ is:

$$\begin{aligned} TV_2 &= p(\lambda)TV_1\log(TV_1) \\ &= p(\lambda)^2\cdot T_p\log(T_p)\cdot\big(\log(p(\lambda)) + \log(T_p) + \log\log(T_p)\big) \\ &\leq p(\lambda)^2\cdot T_p\log(T_p)\cdot\big(3\log(T_p)\big) \\ &\leq 5T_p\cdot\big(p(\lambda)\log(T_p)\big)^2 \end{aligned}$$

  where in the first inequality we used the assumption $T_p \geq p(\lambda)$.

- For $i > 2$, let us assume that $TV_{i-1} = T_p\cdot 5^{i-2}(i-2)!\cdot(p(\lambda)\log(T_p))^{i-1}$. Then the verification

time of $\mathrm{TM}^{(\mathrm{i})}$ is:

$$\begin{aligned}
TV_i &= p(\lambda)TV_{i-1}\log(TV_{i-1}) \\
&\leq T_p \cdot p(\lambda)^i 5^{i-2}(i-2)! \cdot \log(T_p)^{i-1}\big(\log T^{(\mathrm{p})} + \log(5^{i-2}) + \\
&\quad \log((i-2)!) + \log(p(\lambda)^{i-1}) + \log(\log(T_p)^{i-1})\big) \\
&\leq T_p \cdot p(\lambda)^i 5^{i-2}(i-2)! \cdot \log(T_p)^{i-1}(\log T_p + (i-2)\log(5) + \\
&\quad (i-2)\log(i-2) + (i-1)\log(p(\lambda)) + (i-1)\log(\log(T_p))) \\
&\leq T_p \cdot p(\lambda)^i 5^{i-2}(i-2)! \cdot \log(T_p)^{i-1}(\log T_p + (i-1)\log(5) + \\
&\quad (i-1)\log(i-2) + (i-1)\log(p(\lambda)) + (i-1)\log(\log T_p) \\
&\leq T_p \cdot p(\lambda)^i 5^{i-2}(i-2)! \cdot \log(T_p)^{i-1}(5(i-1)\log T_p) \\
&= T_p \cdot p(\lambda)^i 5^{i-1}(i-1)! \cdot \log(T_p)^i
\end{aligned}$$

Above, in the first inequality we applied the inductive hypothesis, in the second we used that, for $i \geq 3$, $(i-2)! \leq (i-2)^{(i-2)}$, in the third step we used $(i-2) \leq (i-1)$, and in the last inequality we used the assumption $T_p \geq p(\lambda), 5, (i-2)$ (namely the result holds asymptotically for such sufficiently large running time T_p).

□

Therefore, the upper bound for TV_t is $5^{t-1}(t-1)!T_p\big(p(\lambda)\log(T_p)\big)^t$. This means that for existing SKHS scheme, our compiler can support a constant number of users.

Comparison with the compiler of [16] For a fair comparison, in B we also analyze the efficiency of the Matrioska compiler [16] using a similar assumption, namely that the SKHS verification circuit has $p(\lambda)S_p \log S_p$ gates, where S_p is the description's size for the input program P that is modeled with the model of [16]. From our analysis in B we get that the size of the verification circuit in the last step is, *at least*

$$S_p \cdot k^t \cdot t! \cdot (t+1)! \cdot p(\lambda)^t (\log p(\lambda))^{2t}$$

which is slightly worse than our upper bound.

6.5.2 General efficiency analysis of our compiler

As we proved in Section 6.5.1, the verification time of the MKHS created using the existing SKHS scheme is exponential. In this section, we examine how the compiler performs in the other cases defined in Section 4.4 and show the cases where our compiler can support a polynomial number of users. We summarize the results of this analysis in Table 2.

Table 2: The verification time in TM-based compiler.

	SKHS verification time	**MKHS verification time**	**number of users**
Case 1	$TV = p(\lambda)T_p \log T_p$	exponential	O(1)
Case 2	$TV = p(\lambda)T_p$	exponential	O(1)
Case 3	$TV = cT_p + np(\lambda)$	exponential	$O(1)$
		quasi-polynomial[a]	$polylog(\lambda)$
		polynomial (if $c = 1$)	$\mathsf{poly}(\lambda)$
Case 4	$TV = S_p \log(T_p) + np(\lambda)$	polynomial	$\mathsf{poly}(\lambda)$
Case 5	$TV = S_p + np(\lambda)$	polynomial	$\mathsf{poly}(\lambda)$

[a] if we can tolerate quasi-polynomial verification cost.

We provide an example of the verification cost for each case in Table 2 and compute the verification cost in the MKHS creates using our compiler.

- **Case 2:** $TV = p(\lambda) \cdot T_p$ for some fixed $p(\lambda) = \mathsf{poly}(\lambda)$.

  In this case, it is easy to see that for all $i = 1$ to t, $TV_{i+1} \leq p(\lambda)^i \cdot T_p$, and thus the running time of $\mathsf{MH.Verif}$ is at most $p(\lambda)^t \cdot T_p$. In this case, the number of users can be constant.

- **Case 3:** $T_p = c \cdot T_p + n \cdot p(\lambda)$ for a constant $c \geq 1$ and a fixed $p(\lambda) = \mathsf{poly}(\lambda)$.

  The verification time of the Turing machine created in step i of our compiler is:

$$TV_{i+1} = c^i \cdot T_p + p(\lambda) \sum_{j=1}^{i} n_j \cdot c^{i-j}$$

  which we prove inductively as follows.

  For $i = 1$, $TV_2 = c \cdot T_p + n_1 p(\lambda)$ holds by construction and by the assumption on TV.

  For any $i \geq 2$, let us assume that

$$TV_i = c^{i-1} \cdot T^{(\mathrm{p})} + p(\lambda) \sum_{j=1}^{i-2} n_j \cdot c^{i-j-1},$$

  then by following the construction we have

$$\begin{aligned} TV_{i+1} &= c \cdot TV_i + n_i p(\lambda) \\ &= c(c^{i-1} \cdot T^{(\mathrm{p})} + p(\lambda) \sum_{j=1}^{i-2} n_j \cdot c^{i-j-1}) + n_i p(\lambda) \\ &= c^i \cdot T^{(\mathrm{p})} + p(\lambda) \sum_{j=1}^{i-1} n_j \cdot c^{i-j}) \end{aligned}$$

  which yields the claimed value. Notice that the value of T can be simplified with an upper bound

$$\begin{aligned} TV_{i+1} &\leq c^i (T_p + p(\lambda) \sum_{j=1}^{i} n_j \\ &\leq c^i (T^{(\mathrm{p})} + i \cdot n \cdot p(\lambda)) \end{aligned}$$

From this bound we deduce that in this case (4) our compiler can support up to $t = O(\log \lambda)$ without incurring in an exponential running time.

- **Case 4:** $TV = S_p \log T_p + np(\lambda)$ for a fixed $p(\lambda) = \mathsf{poly}(\lambda)$.
  According to the size analysis in Section 6.4, we have

$$S_i = S_p - n_i + (|\tau| - 1) \sum_{j=1}^{i-1} n_j + (i-1)(\mathfrak{b}_s(\lambda) + l(\lambda))$$

  We assume for all $i \in [t]$, $S_i \leq S$ where $S = \mathsf{poly}(\lambda)$. Thus, we have

$$TV_{i+1} \leq S_i \log T_i + n_i p(\lambda) \tag{3}$$

  Let $S' = max\{T_p, S\}$. We claim that the verification time in step i of our compiler is:

$$TV_{i+1} \leq 5 \cdot S \cdot \log(S') + n_i p(\lambda)$$

  which we prove inductively as follows.

  For $i = 1$, $TV_2 = S_p \log T_p + n_1 p(\lambda)$ holds by construction and by the assumption on TV, and clearly satisfies the upper bound above.

  For any $i \geq 2$, let us assume that

$$TV_i \leq 5 \cdot S \log(S') + n_{i-1} \cdot p(\lambda)$$

  then we obtain the claimed value from the following sequence of inequalities

$$\begin{aligned} TV_{i+1} &\leq S \log(TV_i) + n_i p(\lambda) \\ &\leq S \log\big(5S \log(S') + n_{i-1} p(\lambda)\big) + n_i p(\lambda) \\ &\leq S\Big(\log 5 + \log S + \log\log(S') + \log(n_{i-1}) + \log(p(\lambda))\Big) + n_i p(\lambda) \\ &\leq 5 \cdot S \log(S') + n_i p(\lambda) \end{aligned}$$

  where the first equality holds by the equation 3, the second inequality holds by applying our inductive hypothesis, and the last one holds by assuming that $S' = max\{T_p, S\}$.

  Since S linearly depends on t, thus S' is a polynomial value based on the security parameter. From this bound we deduce that in this case (4) our compiler can support up to $t = \mathsf{poly}(\lambda)$ users.

- **Case 5:** $TV = S_p + np(\lambda)$ for a fixed $p(\lambda) = \mathsf{poly}(\lambda)$.
  In this case we have

$$TV_{i+1} = S_i + n_i p(\lambda)$$

  Since, S_i linearly depends on the number of users. Thus we can assume, for some $S = poly(\lambda)$, and all $i \in [t]$, $S_i \leq S$. Clearly the verification time in step i of our compiler is:

$$TV_{i+1} \leq S + np(\lambda)$$

  From this bound we deduce that in this case (5), our compiler can support up to $t = \mathsf{poly}(\lambda)$ users.

7 Decreasing the verification cost

The verification time for the existing SKHS construction [21] is $\mathsf{poly}(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})})$ and it depends on the time of the input program. In our compiler, this dependency makes the verification time of MKHS grows exponentially with respect to the number of users (as explained in Section 6.5.1). In this section, we explain how to reduce the verification time of an SKHS with the verification time $\mathsf{poly}(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})})$ to $T^{(\mathrm{p})} + n^{(\mathrm{p})}\mathsf{poly}(\lambda)$.

By considering the efficiency of the delegation system proposed by Kalai et al. [23], we claim that every SKHS which its verification time does not depend linearly to the time of the input program and its input length, can be converted to a new construction such that its verification time linearly depends on the time of the input program. For this conversion, we define the following theorem.

Theorem 2. *Let Π_{HS} be a single-key homomorphic signature scheme that is correct and unforgeable. If Π_{HS} has succinctness l and verification time of $\mathsf{poly}(\lambda)T\log(T)$ where T is the execution time of the input function, and we have an adaptively sound non-interactive delegation scheme Π_{del} with the proof length $L = T^{O(\frac{1}{\log_2(\log_\lambda(T))})}$ and the verification time $O(L)+n\cdot\mathsf{poly}(\lambda)$ where n is the input length, we can build an efficiently verifiable single-key homomorphic signature, which is correct and unforgeable with the succinctness $l + L$ and verification time of $O(L) + n \cdot \mathsf{poly}(\lambda)$.*

For proving this theorem, we first recall the delegation system introduced by Kalai et al. [23] and then, we construct an efficiently verifiable single-key homomorphic signature scheme. Finally we prove the properties described for this scheme.

7.1 Publicly verifiable non-interactive delegation

Let M be a Turing machine with the halting time bound T and input length n. We define U_M as a language consisting of tuples (x, T) such that M accepts x if and only if M halts in atmost T steps where $x \in \{0,1\}^n$ and $n \leq T \leq 2^\lambda$.

Definition 6 (Publicly verifiable non-interactive delegation scheme [23])**.** *A publicly verifiable non-interactive delegation scheme Π_{Del} for the Turing machine M with the time-bound T and the input length n, is a tuple of three probabilistic polynomial-time algorithms* $(\mathrm{Del.S}, \mathrm{Del.P}, \mathrm{Del.V})$*:*

- $(\mathrm{del.pk}, \mathrm{del.vk}) \leftarrow \mathrm{Del.S}(1^\lambda, T, n)$*: the setup algorithm takes the security parameter λ, time-bound T, and input length n as input and outputs the public-key* del.pk*, and the verification-key* del.vk *of the delegation system.*

- $\pi \leftarrow \mathrm{Del.P}(\mathrm{del.pk}, x)$*: the proving algorithm takes the public-key* del.pk *and input message x as inputs, and outputs a proof π. This algorithm is run by the prover.*

- $b := \mathrm{Del.V}(\mathrm{del.vk}, x, \pi)$*: the verification algorithm takes the verification-key* del.vk*, input message x, and proof π as inputs, and outputs a Boolean value. If proof is valid, it outputs one otherwise outputs zero. This algorithm is run by the verifier.*

This delegation system has the following properties:

- ***completeness**: for all λ,n and T we have*

$$pr\left[\mathrm{Del.V}(\mathrm{del.vk}, x, \pi) = 1 \,\middle|\, \begin{array}{l}(\mathrm{del.pk}, \mathrm{del.vk}) \leftarrow \mathrm{Del.S}(1^\lambda, T, n)\\ \pi \leftarrow \mathrm{Del.P}(\mathrm{del.pk}, x)\end{array}\right] = 1$$

- ***soundness**: for all probabilistic polynomial time adversary A and security parameter λ we have*

$$pr\left[\begin{array}{l|l} \text{Del.V}(\text{del.vk}, x, \pi) = 1 & (\text{del.pk}, \text{del.vk}) \leftarrow \text{Del.S}(1^\lambda, T, n) \\ (x, T) \notin U_M & (x, \pi) \leftarrow A(\text{del.pk}, \text{del.vk}) \end{array}\right] \leq negl(1^\lambda)$$

- ***efficiency**: the proof and CRS length is $L = T^{\frac{1}{\log_2(\log_\lambda(T))}}$, the prover's run time is poly(T, λ) and the verifier's run time is $O(L) + n \cdot \mathsf{poly}(\lambda)$.*

7.2 Efficiently verifiable single-key homomorphic signature

Recently, Kalai et al. [23] introduced a publicly verifiable non-interactive delegation system with adaptive soundness for any program with the execution time T which is secure in the CRS model. Their proposed delegation system is efficient, namely, the CRS and the proof length is $L = T^{O(\frac{1}{\log_2(\log_\lambda(T))})}$ and the prover's and verifier's run time are poly(T, λ) and $O(L) + n \cdot \mathsf{poly}(\lambda)$, respectively.

To reduce the verification time in the known SKHS scheme, we use this delegation scheme. In other words, we build an efficiently verifiable single-key homomorphic signature (ESKHS) from the standard SKHS scheme $\Pi_{\mathsf{HS}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ by using the delegation system described in Definition 6. Actually, we use the proof generated by the prover of the delegation system to prove the correct execution of the verification algorithm in the single-key homomorphic signature.

Let $x = ((f, (\tau_1, ..., \tau_n)), \Delta, pk, \sigma', y)$ be the input of the $\mathsf{HS.Verif}$ algorithm and $\hat{M}$ be the Turing description of this algorithm. An efficiently verifiable SKHS is a tuple of five probabilistic polynomial time (PPT) algorithms $\Pi_{\mathsf{EHS}} = (\mathsf{EHS.Setup}, \mathsf{EHS.KeyGen}, \mathsf{EHS.Sign}, \mathsf{EHS.Eval}, \mathsf{EHS.Verif})$ described in Figure 2. As we can see in Figure 2, the length of σ" is $L + l$. So Π_{EHS} satisfies a weaker succinctness property because the signature has a component of length $L = T^{O(\frac{1}{\log_2(\log_\lambda(T))})}$, where T is the runtime of the SKHS verification. Furthermore, the $\mathsf{EHS.Verif}$ algorithm runs Del.V; so, the verification time is $O(L) + |x| \cdot d(\lambda)$ for a fixed polynomial $d(\lambda) = \mathsf{poly}(\lambda)$.

In the case $T = p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})})$, notice that we can bound

$$L = (p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})}))^{\frac{1}{\log_2\log_\lambda(p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})}))}} \leq p(\lambda)(T^{(\mathrm{p})})^{\frac{2}{\log_2\log_\lambda(p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})}))}}$$

which is $\leq p(\lambda)T^{(\mathrm{p})}$ whenever $\log_2\log_\lambda(p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})})) > 2$, which for example holds for $T^{(\mathrm{p})} \geq \lambda^8$.

Correctness Proving the correctness of Π_{EHS} consists of proving the correctness of each signed messages and the correctness of the signature which is the output of the $\mathsf{EHS.Eval}$ algorithm.

- **Authentication correctness**: Authentication correctness is directly derived from the authentication correctness of Π_{sk}.

- **Evaluation correctness**: Since σ" $= (\sigma', \pi)$, σ' is correct based on the evaluation correctness of Π_{sk}, and π is correct based on the completeness of Π_{del}, so Π_{EHS} has the evaluation correctness property.

$\underline{\mathsf{EHS.Setup}(1^\lambda, T^{(\mathrm{v})}, n)}$
- **Run** $\mathsf{pp}' \leftarrow \mathsf{HS.Setup}(1^\lambda)$
- **Run** $(\mathrm{del.pk}, \mathrm{del.vk}) \leftarrow \mathrm{Del.S}(\lambda, , n)$
- **Return** $\mathsf{pp} = (\mathsf{pp}', \mathrm{del.pk}, \mathrm{del.vk})$

$\underline{\mathsf{EHS.KeyGen}(\mathsf{pp})}$
- **Parse** pp **to** $(\mathsf{pp}', \mathrm{del.pk}, \mathrm{del.vk})$
- **Run** $(sk', pk') \leftarrow \mathsf{HS.KeyGen}(\mathsf{pp}')$
- **Return** $\big(sk = sk', pk = pk'\big)$

$\underline{\mathsf{EHS.Sign}(\mathbf{SK}, \Delta, \mathrm{m}, \ell)}$
- **Run** $\sigma \leftarrow \mathsf{HS.Sign}(sk, \Delta, \mathrm{m}, \ell)$
- **Return** σ

$\underline{\mathsf{EHS.Verif}((f, (\tau_1, ..., \tau_n)), \Delta, pk, \sigma", \mathrm{m})}$
- **Parse** $\sigma"$ **to** (σ', π)
- **Run** $b := \mathrm{Del.V}(x, \mathrm{del.vk}, \pi)$
  where $x = (f, pk, \Delta, \sigma', y)$.
- **Return** b

$\underline{\mathsf{EHS.Eval}(f, \Delta, pk, \vec{\sigma})}$
- **Parse** pp **to** $(\mathsf{pp}', \mathrm{del.pk}, \mathrm{del.vk})$
- **Let** $\vec{\mathrm{m}} = (\mathrm{m}_1, ..., \mathrm{m}_n)$
  and $\mathrm{m} = f(\mathrm{m}_1, ..., \mathrm{m}_n)$.
- **Run** $\sigma' \leftarrow \mathsf{HS.Eval}(f, \Delta, pk, \vec{\sigma})$
- **Run** $\pi \leftarrow \mathrm{Del.P}(x, \mathrm{del.pk})$ **where** x **is the inputs of the** $\mathsf{HS.Verif}$ **algorithm which is** $((f, (\tau_1, ..., \tau_n)), \Delta, pk, \sigma', \mathrm{m})$.
- **Return** $\sigma" = (\sigma', \pi)$

Figure 2: Efficiently verifiable SKHS

Security For proving the security of Π_{EHS}, we define the following theorem:

Theorem 3. *If* Π_{HS} *is an unforgeable single-key homomorphic signature and* Π_{del} *is an adaptively sound publicly verifiable non-interactive delegation system under Definition 6, then the efficiently verifiable single-key homomorphic signature scheme* Π_{EHS} *is also unforgeable.*

Proof. Let there exists an adversary A_{EHS} who wins the unforgeability game for the proposed efficiently verifiable single-key homomorphic and outputs a forgery in Π_{EHS}. We show how to construct a new adversary A, that can use Π_{EHS} to do a forgery in Π_{HS} or break the soundness of Π_{del}. Actually, A plays the role of the challenger C in the SKHS unforgeability game for the adversary A_{EHS}.

If the adversary A_{EHS} outputs a forgery ($\mathcal{P}^*$,Δ^*,y^*,$\sigma"^*$) where $\sigma"^* = (sig'^*, \pi^*)$, A can check the validity for the signature σ'^* using $b := \mathsf{HS.Verif}(\mathcal{P}^*, \Delta^*, pk, \sigma'^*, y^*)$ algorithm:

- if $b = 0$, A breaks the soundness of the delegation system and the security of this scheme is reduced to the security of the delegation system.
- if $b = 1$, A does a forgery in the SKHS scheme and the security of this scheme is reduced to the security of the SKHS scheme.

□

Using EHS in our compiler Let us analyze the efficiency of the MKHS scheme that results from using EHS in our compiler as the input SKHS scheme. Also, we consider an instantiation with an SKHS scheme where the verification takes time $T = p(\lambda)T^{(\mathrm{p})}\log(T^{(\mathrm{p})})$, for which we can assume $L \leq p(\lambda)T^{(\mathrm{p})}$.

So, in the EHS signature scheme, we have $C(HSV) = p(\lambda)T^{(\mathrm{p})} + n^{(\mathrm{v})}\mathsf{poly}(\lambda)$ where $n^{(\mathrm{v})}$ is the input size of the verification algorithm. Considering that EHS signatures are not fully succinct,

as $|\sigma"| = |\sigma'| + L$, we have that the input size $n^{(\mathrm{v})}$ of the verification algorithm also depends on T. Essentially, for EHS with $T^{(\mathrm{v})} = \mathsf{poly}(\lambda)T^{(\mathrm{p})}$, which follows our Case 1 in Section 6.5.1, means that with this construction the number of users t needs to be a constant in order to have a polynomial-time verifier.

This leaves the interesting open problem of finding a delegation system, secure in the standard model, in which proofs have better complexity, so that our compiler can yield a more efficient MKHS that can support $O(\log \lambda)$ users.

8 Conclusion

In this paper, we generalized the compiler proposed by Fiore et. al. [16] for converting a single-key homomorphic signature to a multi-key homomorphic signature in any model of computation (not limited to circuits model). We provide a generic efficiency analysis for the compiler. However, this is rather a framework and the full analysis can be obtained only when instantiating the compiler for a specific computational model. We show how Matrioska [16] can be viewed as an instantiation of our compiler in the circuit model and provide an efficiency analysis under tighter assumptions about the complexity of the SKHS verification algorithm. Notably, we show that this instantiation has a growth exponential in the number t of users. Then, we proposed our TM-based instantiation of the general compiler. In particular the technical contribution of this instantiation is the definition of the multi-tape TM model and the design of the Mask function that works efficiently on it. Then, we analyze the efficiency of the TM-based instantiation. Notably, we show that some assumptions our compiler yields a MKHS in which the growth is linear in the number of users and thus it can tolerate a polynomial number of users. Since the verification time of the SKHS scheme affects on the verification time of the MKHS scheme created by our compiler, we provide a transformation to improve the efficiency of the verification algorithm in the SKHS scheme using a delegation system. With state-of-the-art delegation schemes that are secure in the standard model (i.e., no random oracles) under falsifiable assumptions, we obtain an SKHS scheme whose verification is $\mathsf{poly}(\lambda)^t T^{(\mathrm{p})}$. We leave it as an open problem to design an SKHS or a delegation scheme with better succinctness, which would directly allow our MKHS compiler to have a polynomial verification time, and thus support an arbitrary polynomial number of users.

Acknowledgements

We would like to thank the anonymous reviewers of this article for the useful comments and suggestions that allowed us to improve our work.

This work has received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program under the project PICOCRYPT (grant agreement No. 101001283), by the Spanish Government under the projects SCUM (ref. RTI2018-102043-B-I00), CRYPTOEPIC (ref. EUR2019-103816), and SECURITAS (ref. RED2018-102321-T), the Madrid Regional Government under the project BLOQUES (ref. S2018/TCS-4339), and by Sapienza University of Rome under the grant SPECTRA.

Reference

[1] S. Agrawal, D. Boneh, X. Boyen, and D. M. Freeman. Preventing pollution attacks in multi-source network coding. In P. Q. Nguyen and D. Pointcheval, editors, *Public Key Cryptography – PKC 2010*, pages 161–176, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] D. F. Aranha and E. Pagnin. The simplest multi-key linearly homomorphic signature scheme. In P. Schwabe and N. Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 280–300, Cham, 2019. Springer International Publishing.

[3] N. Attrapadung and B. Libert. Homomorphic network coding signatures in the standard model. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 17–34. Springer, Heidelberg, Mar. 2011.

[4] N. Attrapadung, B. Libert, and T. Peters. Computing on authenticated data: New privacy definitions and constructions. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 367–385. Springer, Heidelberg, Dec. 2012.

[5] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 863–874, New York, NY, USA, 2013. Association for Computing Machinery.

[6] D. Boneh, D. Freeman, J. Katz, and B. Waters. Signing a linear subspace: Signature schemes for network coding. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 68–87, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In K. G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 149–168, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] D. Boneh and D. M. Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, editors, *Public Key Cryptography – PKC 2011*, pages 1–16, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] D. Catalano, D. Fiore, R. Gennaro, and K. Vamvourellis. Algebraic (trapdoor) one-way functions and their applications. In A. Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 680–699. Springer, Heidelberg, Mar. 2013.

[10] D. Catalano, D. Fiore, and L. Nizzardo. Programmable hash functions go private: Constructions and applications to (homomorphic) signatures with shorter public keys. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 254–274. Springer, Heidelberg, Aug. 2015.

[11] D. Catalano, D. Fiore, and B. Warinschi. Adaptive pseudo-free groups and applications. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 207–223. Springer, Heidelberg, May 2011.

[12] D. Catalano, D. Fiore, and B. Warinschi. Efficient network coding signatures in the standard model. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *Public Key Cryptography – PKC 2012*, pages 680–696, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] D. Catalano, D. Fiore, and B. Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 371–389, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[14] D. Catalano, A. Marcedone, and O. Puglisi. Authenticating computation on groups: New homomorphic primitives and applications. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 193–212. Springer, Heidelberg, Dec. 2014.

[15] D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin. Multi-key homomorphic authenticators. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 499–530, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[16] D. Fiore and E. Pagnin. Matrioska: A compiler for multi-key homomorphic signatures. In D. Catalano and R. De Prisco, editors, *Security and Cryptography for Networks*, pages 43–62, Cham, 2018. Springer International Publishing.

[17] M. Fischer and N. Pippenger. Mj fischer lectures on network complexity. *Universit/it Frankfurt*, 1974.

[18] D. M. Freeman. Improved security for linearly homomorphic signatures: A generic framework. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 697–714. Springer, Heidelberg, May 2012.

[19] R. Gennaro and D. Wichs. Fully homomorphic message authenticators. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 301–320, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 99–108, New York, NY, USA, 2011. Association for Computing Machinery.

[21] S. Gorbunov, V. Vaikuntanathan, and D. Wichs. Leveled fully homomorphic signatures from standard lattices. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 469–477, New York, NY, USA, 2015. ACM.

[22] R. Johnson, L. Walsh, and M. Lamb. Homomorphic signatures for digital photographs. In G. Danezis, editor, *Financial Cryptography and Data Security*, pages 141–157, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] Y. T. Kalai, O. Paneth, and L. Yang. How to delegate computations publicly. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 1115–1124, New York, NY, USA, 2019. ACM.

[24] R. W. F. Lai, R. K. H. Tai, H. W. H. Wong, and S. S. M. Chow. Multi-key homomorphic

signatures unforgeable under insider corruption. In T. Peyrin and S. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 465–492, Cham, 2018. Springer International Publishing.

[25] B. Libert, T. Peters, M. Joye, and M. Yung. Linearly homomorphic structure-preserving signatures and their applications. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 289–307. Springer, Heidelberg, Aug. 2013.

[26] L. Schabhüser, D. Butin, and J. Buchmann. Context hiding multi-key linearly homomorphic authenticators. In M. Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 493–513, Cham, 2019. Springer International Publishing.

[27] M. Sipser. Introduction to the theory of computation. *SIGACT News*, 27(1):27–29, Mar. 1996.

A Proof of the correctness

We prove the correctness of MKHS scheme generated by the compiler by defining the following two lemmas.

Lemma 1 (Authentication correctness)**.** *Let* $\Pi_{\mathsf{HS}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ *be an SKHS scheme that has the authentication correctness property, then the MKHS scheme* $\Pi_{\mathsf{MH}} = (\mathsf{MH.Setup}, \mathsf{MH.KeyGen}, \mathsf{MH.Sign}, \mathsf{MH.Eval}, \mathsf{MH.Verif})$ *obtained from the compiler has the authentication correctness property according to the correctness of the scheme defined in Definition 1.*

Proof. Let the $\mathsf{MH.Setup}(1^\lambda, N_t, cp)$ algorithm generates pp and $\mathsf{MH.KeyGen}(\mathsf{pp})$ algorithm generates sk_{id} and pk_{id} for the user with identity $id \in \mathcal{ID}$. For any $\mathrm{m} \in \mathcal{M}$, any dataset identifier $\Delta \in \{0,1\}^*$, and any $\ell = (id, \tau) \in \mathcal{L}$, if σ is the output of $\mathsf{MH.Sign}(sk_{id}, \Delta, \mathrm{m}, \ell)$, we should show that we have $\mathsf{MH.Verif}(\mathcal{I}, \Delta, pk_{id}, \sigma, \mathrm{m}) = 1$, where $\mathcal{I} = (I, \ell)$ is an identity function such that $I(\mathrm{m}) = \mathrm{m}$. In our compiler, $\mathsf{MH.Sign}(sk_{id}, \Delta, \mathrm{m}, \ell = (id, \tau))$ runs the $\sigma \leftarrow \mathsf{HS.Sign}(sk_{id}, \Delta, \mathrm{m}, \tau)$ algorithm. For verifying the output of the identity program $\mathcal{I}$, since $t = n = 1$, the compiler runs $\mathsf{HS.Verif}\big((I, (\tau)), \Delta, pk_{id}, \sigma, \mathrm{m}\big)$. So, If Π_{HS} has the authentication correctness property, Π_{MH} also has the authentication correctness property. □

Lemma 2 (Evaluation correctness)**.** *Let* $\Pi_{\mathsf{HS}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ *is an SKHS scheme that has the evaluation correctness property, then the MKHS scheme* $\Pi_{\mathsf{MH}} = (\mathsf{HS.Setup}, \mathsf{HS.KeyGen}, \mathsf{HS.Sign}, \mathsf{HS.Eval}, \mathsf{HS.Verif})$ *obtained from the compiler has the evaluation correctness property according to the correctness of scheme defined in Definition 1.*

Proof. For proving the evaluation correctness, we show that the output of computation on the signed messages which are signed correctly, should verified correctly. If $(t = 1)$, the compiler works like the SKHS scheme. So, if Π_{sk} has the evaluation correctness property, Π_{MH} also has the evaluation correctness property. If $(t > 1)$, we prove the evaluation correctness property using induction. Let E_0 is the function created by running the $\mathsf{Compose}$ function on F to get a single output function.

- For the first user, the $\mathsf{HS.Eval}$ algorithm runs on E_1 and generates σ'_1. Verification of σ'_1 is done using $\mathsf{HS.Verif}$ algorithm which is run on E_1. If $E_1(m_1, ..., m_n) = 1$, then $\mathsf{HS.Verif}$

outputs one. Since E_1 takes its inputs from the first user, if Π_{HS} has the evaluation correctness property, then σ'_1 is verified correctly.

- In the i-th step in the compiler where $i \in [t]$, the $\mathsf{MH.Eval}$ algorithm, creates a new function E_i which works on inputs from the user with identity id_i. Let $s_i = \sum_{j=1}^{i-1} n_j$ and $e_i = \sum_{j=1}^{i} n_j$. Each of this functions is evaluated using the evaluation algorithm as below.

$$\sigma'_i \leftarrow \mathsf{HS.Eval}\Big(\big(E_i, (\tau_{s_i}, ..., \tau_{e_i})\big), \Delta, pk_i, \{\sigma_{s_i}, ..., \sigma_{e_i}\}\Big)$$

For proving the evaluation correctness using induction, we assume σ'_{t-1} is verified correctly, then we should show that σ'_t is verified correctly, too. In the verifier side, using the third property of Mask, E_t is created using the function Mask^*. The other way to create E_t is by running the mask function on HSV_t which is the description of the function corresponding to the $\mathsf{HS.Verif}\Big(\big(E_{t-1}, (\tau_{s_{t-1}}, ..., \tau_{e_{t-1}})\big), \Delta, pk_{t-1}, \sigma'_{t-1}, 1\Big)$ algorithm. By the property I of Mask, the function E_t will be such that

$$E_t(m_{s_t}, ..., m_{e_t}) = \begin{cases} 1 & \text{if } \mathsf{HS.Verif}\Big(\big(E_{t-1}(m_{s_{t-1}}, ..., m_{e_{t-1}}), (\tau_{s_{t-1}}, ..., \tau_{e_{t-1}})\big), \\ & \quad \Delta, pk_{t-1}, \sigma'_{t-1}, 1\Big) = 1 \\ 0 & \text{otherwise} \end{cases},$$

Since we assume σ'_{t-1} is the correct signature for the output of E_{t-1}, then the output of E_t will be one. Since one is the correct output of E_t and Π_{HS} has the evaluation correctness property, thus σ'_t is verified correctly. So all the signatures $(\sigma'_1, ..., \sigma'_t)$ are verified correctly and Π_{MH} has the evaluation correctness property.

□

B Circuit size growth in Matrioska for the existing SKHS scheme

Let S determines the size of the circuit C that models the program p. We assume SKHS verification circuit has $q_{HSV} = p(\lambda)S \log S$ gates for a fixed $p(\lambda) = \mathsf{poly}(\lambda)$ and investigate the growth in the circuit size.

Let the description size of the circuit E_i is denoted by S_i. As explained in [16], the description size of the circuit $E_i = (n_i, o_i, q_i, L_i, R_i, G_i)$ is computed as $S_i = (n_i + q_i)\log(n_i + q_i)$. Let $HSV_i = (n_{HSV_i}, 1, q_{HSV_i}, L_{HSV_i}, R_{HSV_i}, G_{HSV_i})$ be the description of a circuit that models the verification algorithm of the SKHS scheme that is created in the i-th step of the compiler. This circuit is supposed to be run on inputs $\Big(\big(E_{i-1}, (\tau_{1+\sum_{j=1}^{i-1} n_j}, ..., \tau_{\sum_{j=1}^{i} n_j})\big), \Delta, pk_{i-1}, \sigma'_{i-1}, 1\Big)$. For simplicity we assume that $l(\lambda) = |\Delta| + |pk| + |\sigma'| + 1$.

In this case, for every $1 \le i \le t$, we have that $q_{HSV_i} = p(\lambda)S_{i-1}\log(S_{i-1})$. Let E_0 be a circuit with q_0 gates; hence $S_0 = kq_0 \log q_0$. We claim that

$$S_i \ge \begin{cases} kq_0 \log q_0 & i = 1 \\ k^{i+1} \cdot i! \cdot (i+1)! \cdot p(\lambda)^i q_0 (\log p(\lambda))^{2i+1} & 2 \le i \le t \end{cases}$$

- For $i = 1$, we know $E_1 = \mathsf{Compose}(M_1, E_0)$, by using our assumption on E_0, we get $q_{E_1} = n + q_0$. Hence

$$S_1 \geq k \cdot q_0 \log(q_0)$$

For $2 \leq i \leq t$ we prove the claim by induction.

- For $i = 2$, we know $E_2 = \mathsf{Compose}(M_2, HSV_2)$. Hence

$$\begin{aligned}
S_2 &\geq Size(HSV_2) = kq_{HSV_2} \log(q_{HSV_2}) \\
&\geq kp(\lambda) \cdot S_1 \log(S_1) \cdot \log(p(\lambda) S_1 \log(S_1)) \\
&\geq k^3 \cdot 2 \cdot p(\lambda)^2 q_0 (\log p(\lambda))^3 \cdot \log(2k^2 p(\lambda) q_0 (\log p(\lambda))^3) \cdot \\
&\quad \log(p(\lambda)^2 2k^2 q_0 (\log p(\lambda))^3) \\
&\geq k^3 \cdot 2 \cdot p(\lambda)^2 q_0 (\log p(\lambda))^3 \cdot (\log p(\lambda) + \log q_0) \cdot (2 \log p(\lambda) + \log q_0) \\
&\geq k^3 \cdot 2 \cdot p(\lambda)^2 q_0 (\log p(\lambda))^3 \cdot 2 \log(p(\lambda)) \cdot 3 \log(p(\lambda)) \\
&\geq k^3 \cdot 2! \cdot 3! \cdot p(\lambda)^2 q_0 (\log p(\lambda))^5
\end{aligned}$$

where in the first inequality we used our assumption on $q_{HSV_i} = p(\lambda) S_{i-1} \log(S_{i-1})$; in the second inequality we used the bound on S_1 shown above (with a simplification); and in the fourth inequality we used the assumption that $q_0 \geq p(\lambda)$.

- For $i > 2$, let us assume that

$$S_{i-1} \geq k^i \cdot i! \cdot (i-1)! \cdot p(\lambda)^{i-1} q_0 (\log p(\lambda))^{2i-1}$$

Then we have

$$\begin{aligned}
S_1 &\geq Size(\mathsf{Compose}(M_i, HSV_i)) \\
&\geq Size(HSV_i) = kq_{HSV_i} \log(q_{HSV_i}) \\
&\geq kp(\lambda) \cdot S_{i-1} \log(S_{i-1}) \cdot \log(p(\lambda) S_{i-1} \log(S_{i-1})) \\
&\geq k^{i+1} \cdot i! \cdot (i-1)! \cdot p(\lambda)^i q_0 (\log p(\lambda))^{2i-1} \cdot \\
&\quad \log(k^i \cdot i! \cdot (i-1)! \cdot p(\lambda)^{i-1} q_0 (\log p(\lambda))^{2i-1}) \cdot \\
&\quad \log(p(\lambda) k^i \cdot i! \cdot (i-1)! \cdot p(\lambda)^{i-1} q_0 (\log p(\lambda))^{2i-1}) \\
&\geq k^{i+1} \cdot i! \cdot (i-1)! \cdot p(\lambda)^i q_0 (\log p(\lambda))^{2i-1} \cdot \\
&\quad ((i-1) \log p(\lambda) + \log q_0) \cdot \\
&\quad (\log p(\lambda) + (i-1) \log p(\lambda) + \log q_0) \\
&\geq k^{i+1} \cdot i! \cdot (i-1)! \cdot p(\lambda)^i q_0 (\log p(\lambda))^{2i-1} \cdot i \log p(\lambda) \cdot (i+1) \log p(\lambda) \\
&= k^{i+1} \cdot i! \cdot (i+1)! \cdot p(\lambda)^i q_0 (\log p(\lambda))^{2i+1}
\end{aligned}$$

where in the first inequality we used our assumption on $q_{HSV_i} = p(\lambda)\ S_{i-1} \log(S_{i-1})$; in the second inequality we used the bound on S_{i-1} from the inductive assumption; and in the fourth inequality we used the assumption that $q_0 \geq p(\lambda)$.