Public Verifiable Function Secret Sharing

Q. Wang[1], F. Zhou[1] C. Chen[1], F. Li[1], and Z. Xu[1]

Northeastern University, Shenyang NO. 3-11, Wenhua Road, Heping District, China,
wangq3635@126.com

Abstract. Function secret sharing(FSS) was introduced by Boyle et al. in Eurocrypt 2015, which allowed a dealer to split a secret function f into n separate pieces such that each piece enables the server who owns it to generate a secret share of the evaluation of $f(x)$. However, when just only one collusive server returns a wrong result, reconstructing the secret will fail. Therefore, we are required to find an applicable approach to check the correctness of result returned by the untrusted server. To solve this issue, we firstly introduce a primitive called Public Verifiable Function Secret Sharing ($\mathcal{PVFSS}$), and define three new important properties: public delegation, public verification and high efficiency. Then we initiate a systematic study of $\mathcal{PVFSS}$ and construct a $\mathcal{PVFSS}$ scheme for point function. Not only captures our scheme these three properties, but also allows the client to verify the outcome in time constant i.e., in indeed substantially less time than performing the computation locally. We conducted a performance analysis, which manifested that our scheme can be applied into practice such as cloud/outsource computing.

Keywords: PVFSS, high efficiency, public delegation, public verification

1 Introduction

Previous researches on secret sharing scheme(SSS) [1,2,3,4,5] put lots of focus on a real number rather than a function. These schemes allow one to split a secret(a real number) s into n pieces and distribute them to participants respectively, such that only universal set of the pieces can reconstruct it. However, as a matter of fact, a function would have comparatively higher theoretical and practical value than a point (a real number). Motivated by this, if only the secret was a function, we would also deploy the SSS to cloud computing [6,7,8,9] to request a powerful server to perform some heavy computations of a sub-function about the secret on arbitrary input, such as a stock manipulation or cryptographic operator. Finally, the server returns the results to a client with low computational ability. Not until [10] implemented the Function Secret Sharing does this hot potato seem to be worked out. At a high level, a p-party FSS scheme for a class $\mathcal{F}$ of functions f allows a dealer to split an arbitrary sharing $f \in \mathcal{F}$ into p sub-functions $f_i : \{0,1\}^n \to \mathbb{G}$,$1 \leq i \leq p$,such that (1) $\sum_{i=1}^{p} f_i(x) = f(x)$, (2) each f_i is described by a short key k_i and (3) any strict subset of the f_i hides f, i.e.

there doesn't exist any key that can individually reveal information about which function f has been shared.

However, once even only or no more than one malicious server outputs a malicious or wrong result, the recovery may break down. Therefore, it is imperative to propose a mechanism or approach, in the perspective of security, to enforce the integrity and correctness of the computations. This incentive triggers an enormous challenge on prior secret sharing schemes. To deal with this great challenge, chor et al. [11] introduced the cryptographic primitive of verifiable secret sharing (VSS), where everyone can verify whether all received pieces is valid or not so that only correct result will be received by the client. Similarly, this concept is designed to verify a point rather than a function. Therefore, it is impossible to use the method of [11] to settle this matter.

To solve the problems above and achieve this wishful thinking, to the best of our knowledge, we firstly present the primitive called public verifiable function secret sharing. Our concept is a (somewhat surprising) connection between the notions of Function Secret Sharing (FSS) and verifiable computation [12,13,14,38]. Moreover, we propose three new properties as following:

-Public Delegation: everyone could be able to delegate computations to powerful cloud servers, which means that a dealer desires to outsource the computation so that he or anybody can supply inputs to the function and perform the delegation over and over again.

-Public Verification: the dealer needs to produce a public verifiable key, which endows the client (verifier) the privilege to check the integrity and correctness of the computation among untrusted servers. In other word, the wrong result would never be received.

-High Efficiency: the verification of such correctness proof must cost substantially less effort than the computation, otherwise the client would either not be able to verify the proof, or execute computation by himself.

1.1 Our Results and Contributions

In this paper, we extend the work of [10] and propose a new paradigm called $\mathcal{PVFSS}$ to check the correctness and integrity of the results returned by untrusted servers. We now consider the following scenario: allow a dealer to divide a secret function into several sub-functions. Each sub-function is distributed or outsourced to the server. On issuing a query, each server outputs evaluation of their own sub-function, then the client can reconstruct the secret. However, once no more than or at least one collusive server performs the procedure maliciously, then a wrong result would be shared to the client. One of our technical highlights is a new method that allows any client to verify whether the server returns a malicious or wrong result.

Our major results and contributions are summarized as follows.

Definition of New Primitive. We are the first ones to formally define public verifiable function secret sharing ($\mathcal{PVFSS}$) and its security.

Public Delegation. Our construction allows a dealer to divide the secret function among lots of servers. Then arbitrary clients could be able to submit inputs for delegation.

Public Verification. Our construction allows arbitrary clients and not just the dealer to check the integrity and correctness of the computation. Namely, a wrong result would never be received by the client.

High Efficiency. Our construction allows a computationally weak client to verify the correctness of secret function in constant time. That costs substantially less effort than performing a computation from scratch, otherwise the client would either not be able to verify the proof, or execute the computation by himself.

Ingenious Scheme for Point Function. Our construction for point function presents three types of results. First, we prove theoretically the existence that two different hash functions H_y, H_r can map to an overall hash function H_w by algebraic operation. Second, an approach was introduced to construct a hash function H_w satisfying $H_w = aH_y + H_r$. Third, our verifiable scheme captures these three properties. Meanwhile, not only key size but also each algorithm efficiency costs twice less effort than FSS's.

Contrastive Analysis of $\mathcal{PVFSS}$ and FSS. The key length of $\mathcal{PVFSS}$ is $O(p^{\mu} \cdot p^{n-1}(\lambda \cdot +2\,|y|))$, which is far less than FSS's $O(2 \cdot p^{\mu} \cdot p^{n-1}(\lambda \cdot +|y|))$. A comparative experiment was conducted on $\mathcal{PVFSS}$ and FSS. The results of the experiments manifest that $\mathcal{PVFSS}$'s performance is similar to FSS's.

Contribution to Cloud or Outsource Computing. A dealer holds a secret function and hopes to share it among several clients without revealing any information about the secret function f . However, the client could get the evaluation $f(x)$. The property, Public verification, was added to FSS so that we can deploy improved FSS to cloud computing. Our scheme allows the dealer to execute an expensive splitting operation. Then the sub-function is distributed or outsourced to the powerful but untrusted server. On issuing a query, the untrusted server outputs a result. Finally, a client with a weak computational ability can verify the result in a constant time. That satisfies the cost of verification is far less than that of computation locally.

1.2 Related Work

Threshold cryptography [15] deals with the problem of sharing a highly sensitive secret (a point) among a group of users so that only a sufficient number of them gathering together can recover the secret. Typically, some classic methods, reconstructing secret sharing, include polynomial interpolation [1], hyperplane geometry [16], Chinese Remainder Theorem (CRT) [17]. All of these schemes assumed that the dealer and sharers are absolutely reliable, however, there is always some deviation in reality that server may misbehave and distribute inconsistent shares to participants or some hardware failures occurred. In this scenario, participants will unable to reconstruct the secret. Not until the B. Chor et al. [11] proposed Verifiable Secret Sharing, based on Shamir's SSS, did this issue be addressed. [11] achieve a significant property that not only the participants, but

especially everybody can verify that the shares have been correctly distributed. Markus Stadler [18] calls such schemes publicly verifiable secret sharing (PVSS) schemes. These schemes have been extensively studied and used in threshold cryptography and secure multiparty computation [19,20,36,37].

In the perspectives of authentication or verifiability, we consider that Fault-Tolerant Key Agreement (FTKA) [21,22,33,34,35] belongs to PVSS for the reason that the honest participants can agree on a common key even though there exists some malicious participants, i.e., the honest can be able to check the integrity and correctness of the sharing secret (key). In 2015, [10] introduces the new notion called FSS, which is a generalization of distributed point functions (DPF) [23]. Amazingly, [10] achieves function secret which has comparatively higher theoretical and practical value than the real number. We can deploy the FSS to cloud computing to request a powerful server to perform some heavy computation of a sub-function about the secret on arbitrary input. Finally, the server returns the results to a computationally weak client. Inspired by that, we come up with an idea that whether we could use the method of [11] to settle the above matter in FSS. In other words, add a verifiable property to FSS. This may sound like a vagarious thinking or in general impossible just because the secret is a point rather than a function. So some difficulties still should be overcome. Anyway, we must have to settle the verifiable problem.

Organization. The rest of this paper is organized as follows. Section 2 recalls some knowledge needed in our scheme beforehand. Section 3 formally defines a new primitive called $\mathcal{PVFSS}$, describes the protocol, and gives the correctness and security definition needed by our protocol, respectively. Section 4 provides details of construction of the proposed scheme, correctness and security proof, and analyzes theoretically the key length and each algorithm efficiency, respectively. Furthermore, lots of comparative experiments are conducted between FSS and $\mathcal{PVFSS}$. Section 5 introduces some applications about our scheme. Finally, section 6 makes a conclusion and extends some interesting and challenging open problems.

2 Preliminaries

Definition 1. (Hash Function). *Hash Function $h : M \to R$ is a map that can be used to transform data of arbitrary size to data of fixed size. The hash value is representative of the original string of characters, but is normally smaller than the original. At a minimum, a secure hash function must have the following properties:*

-Pre-Image Resistance: *Given a hash value y should be difficult to find any message m such that $y = h(m)$.*

-Second Pre-Image Resistance: *Given an input m_1 it should be difficult to find another input m_2 such that $h(m_1) = h(m_2)$.*

-Collision Resistance: *It should be difficult to find two different messages m_1 and m_2 such that $h(m_1) = h(m_2)$.*

Definition 2. (Point function). *Point function is a function about a special point* (x, y)*. For* $x, y \in \{0,1\}^*$*, the point function is defined as:*

$$P_{x,y}(\hat{x}) = \begin{cases} y & if\ \hat{x} = x \\ \\ 0 & if\ \hat{x} \neq x \end{cases}$$

Definition 3. (Output Decoder). *Output decoder is a decoding procedure DEC, which is a tuple* $(S_1, \cdots, S_n, R, Dec)$ *satisfying: share spaces* $S_1, \cdots, S_n$ *among* n *parties; output space* R*; and a decoder function* $Dec : S_1 \times \cdots \times S_n \to R$ *taking* n *shares to an output over an Abelian group structure.*

Definition 4. (Function Secret Sharing). *A FSS scheme (Function Secret Sharing) for a function family* F *is a tuple of triple PPT algorithms with the following specification:*

1. $(\mathrm{EK}_1, \cdots, \mathrm{EK}_\mathrm{n}) \leftarrow \mathrm{FSS.KeyGen}(1^\lambda, f, n)$*: Algorithm* KeyGen *takes as input the security parameter* λ*, the number of sharers* n *and a function* f *belonging to a function family* F*. It outputs key pairs* $(\mathrm{EK}_1, \cdots, \mathrm{EK}_\mathrm{n})$*.*

2. $y_i \leftarrow \mathrm{FSS.Eval}(i, EK_i, x)$*: Algorithm* Eval *takes as input the a party index* i*, key* k_i *and input string* x*. It outputs a value* $y_i \in S_i$ *for the function* $f(x)$*.*

3. $y \leftarrow \mathrm{FSS.Dec}(y_i, \cdots, y_n)$*: Algorithm* Dec *takes as input the* n *parties' shares* $(y_i, \cdots, y_n)$*. It outputs a value* $y \in R$*, corresponding to the function* $f(x)$*.*

For full version and concrete construction, we refer the reader to the full version of this paper [10].

3 $\mathcal{PVFSS}$ Definition

In section 2, we describe a function secret sharing scheme, which splits the function secret into some sub-functions. Any strict subset of them hide the secret. Meanwhile, an adversary can't get any information about the secret function. However, we assuredly have to solve security problem in that scheme, if we can't promise that the server (sharer) does not give a malicious sharing piece. Namely, how to guarantee that the client surely receives a correct secret. Motivated by verifiable secret sharing, we firstly upgrade their definition and construct a scheme by adding verification that checks the integrity and correctness of the function secret sharing scheme, and call it Public Verifiable Function Secret Sharing. Moreover, we propose three new properties of public verifiable function secret sharing schemes, namely

Public Delegation. Our construction allows a dealer to divide the secret function among lots of servers. Then arbitrary clients could be able to submit inputs for delegation.

Public Verification. Our construction allows arbitrary clients and not just the dealer to check the integrity and correctness of the computation. Namely, a wrong result would never be received by the client.

High Efficiency. Our construction allows a computationally weak client to verify the correctness of secret function in constant time. That costs substantially less effort than performing a computation from scratch, otherwise the client would either not be able to verify the proof, or would execute the computation by himself.

Together, a function secret sharing protocol that satisfies all properties is called a public verifiable function secret sharing protocol. The following definition captures these three properties.

3.1 Model Definition

We now formally define our notion of a Public Verifiable Function Secret Sharing scheme $\mathcal{PVFSS} = (\text{Setup}, \text{KeyGen}, \text{Eval}, \text{Dec}, \text{Verify})$ consisting of the following algorithms:

$(SK_f, VK_f) \leftarrow \text{Setup}(1^\lambda, f)$: The Setup algorithm takes as input security parameter λ and secret function f. It outputs a secret key SK_f and a public verifiable key VK_f.

$(EK_1, \cdots, EK_n) \leftarrow \text{KeyGen}(f, n, SK_f)$: The key generation algorithm takes as the target function f, the number of sharers n and a secret key SK_f. It generates n evaluation keys $(EK_1, \cdots, EK_n)$, which are respectively represented sub-functions $(f_1, \cdots, f_n)$.

$(f_i(x), \omega_{f_i(x)}) \leftarrow \text{Eval}(i, EK_i, x)$: The evaluation algorithm takes as input a server index i, a evaluation key EK_i and input string x. It returns a value $f_i(x)$ and a witness $\omega_{f_i(x)}$ both at input x.

$(y, \omega_{f(x)}) \leftarrow \text{Dec}(y_1, \omega_{f_1(x)}, \cdots, y_n, \omega_{f_n(x)})$: The decoding algorithm takes as input the n servers' results pair $(y_i, \omega_{f_i(x)})$. It outputs the evaluation y, where $y = f(x)$ is the output of the function f at input x, together with a witness $\omega_f(x)$ at point x.

1 *or* $\perp \leftarrow \text{Verify}(\omega_{f(x)}, y, VK_f, x)$: The verification algorithm takes as input the witness $\omega_{f(x)}$, the result y and the public verifiable key VK_f. It outputs either $\perp$ or 1. Here, the special symbol $\perp$ signifies that the verification algorithm rejects the answer . Otherwise, it accepts the evaluation y.

Protocols. In this section, we briefly describe how the above algorithms of a public verifiable function secret sharing scheme are applied in a three-party protocol, including a trusted dealer, an untrusted server and a client. Initially, the dealer owns a function f and wants to share it with client without revealing any information about the function. The dealer cut the secret into some slices, each of which is outsourced to one untrusted server. On issuing a query, the server returns a result together with a witness so that the client can leverage the witness to check the correctness of the computation. The dealer performs algorithm Setup, which takes as input security parameter λ and a secret function f, generates a secret key SK_f about the secret function f. Meanwhilealgorithm Setup also gives a public verifiable key VK_f so that all of the clients will be able to

verify the correctness of the results. Similarly, he runs algorithm KeyGen to pick the keys $(EK_1, \cdots, EK_n)$, which is respectively distributed to the *i-th* server for all $i \in [n]$. Then, the client issues a query x on all servers. On an input query x, each server computes and replies with $(f_i(x), \omega_{f_i(x)}) \leftarrow \text{Eval}(i, EK_i, x)$. As soon as the client receives all the result pairs $(f_i(x), \omega_{f_i(x)})$ from 1 to n, he would decode and reconstruct result y, same as the witness $\omega_f(x)$, by using algorithm Dec. Finally, the client executes algorithm Verify, which takes $\omega_{f(x)}, y, VK_f, x$ as input, in order to verify the correctness of result y. The result y is accepted if and only if $1 \leftarrow \text{Verify}(\omega_{f(x)}, y, VK_f, x)$. Otherwise, the client rejects the result y.

3.2 Correctness and Security Definitions

In this section we give necessary definitions that will be used in the rest of the paper. Let λ be a security parameter. We say that a function $\text{neg}(\lambda)$ is a negligible function of λ, if $\text{neg}(\lambda)$ less than $1/p(\lambda)$, for all polynomials $p(\lambda)$. PPT stands for *probabilistic polynomial time* and $\mathcal{A}(\cdot)$ stands for a probabilistic polynomial time (PPT) algorithm. Given $n \in N$, let E_n and O_n denote subsets of binary arrays of size $n \times 2^{n-1}$. Let $E_n(O_n)$ denote the set of all arrays such that the columns of each array are all n-bits strings with an even(odd) number of 1 bits.

A public verifiable function secret sharing scheme should be correct, secure and privacy. Intuitively, this scheme is correct if whenever its algorithms are executed honestly, it never rejects a correct answer. More formally:

Definition 5. ($\mathcal{PVFSS}$ Correctness). *A Public Verifiable Function Secret Sharing is correct. For any function $f \in F$, it runs the algorithm* Setup *to generate a secret key SK_f and a public verifiable key VK_f. The key generation algorithm produces n keys $(EK_1, \cdots, EK_n) \leftarrow \text{KeyGen}(\text{f}, \text{n}, \text{SK}_\text{f})$ such that, for any $x \in Domain(f)$, if, any $(f_i(x), \omega_{f_i(x)}) \leftarrow \text{Eval}(i, EK_i, x)$ for all $i = 1, \cdots, n$, and $(y, \omega_f(x)) \leftarrow \text{Dec}(\delta_{f_1(x)}, \delta_{\omega_{f_1(x)}} \cdots, \delta_{f_n(x)}, \delta_{\omega_{f_n(x)}})$ then $1 \leftarrow \text{Verify}(\omega_x, y, SK_f)$.*

Intuitively, verifiable function secret sharing is secure, which contains two cases. One case is t-security that the adversary can't tell the difference between the two secrets, except with negligible probability. It is noted that t is smaller than n. Another one is full security that the adversary cannot persuade a verifier to accept a wrong computational result. More formally:

Definition 6. ($\mathcal{PVFSS}$ t-Security). *Let $\mathcal{PVFSS}$ be a public verifiable function secret sharing scheme for a function $f \in F$ among n parties, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1)$ be a two-tuple of probabilistic polynomial-time machines. We define security via the following experiment.*

$$
\begin{aligned}
&\textit{Experiment } \mathbf{EXP}_{\mathcal{A}}^{t-Sec}[\mathcal{PVFSS}, F, n, \lambda]: \\
&\quad (f_0, f_1, \cdots) \leftarrow \mathcal{A}_0(1^\lambda); \\
&b \in_R \{0,1\}; \\
&(SK_{f_b}, VK_{f_b}) \leftarrow \text{Setup}(1^\lambda, f_b); \\
&(EK_1, \cdots, EK_n) \leftarrow \text{keyGen}(f_b, n, SK_{f_b}); \\
&\quad \hat{b} \leftarrow \mathcal{A}_1((k_i)_{i \in T}); \\
&If\ b = \hat{b}, output\ '1',\ else\ '0'
\end{aligned}
$$

The experiment is valid if $f_0, f_1 \in F$. Based on a typical indistinguishability argument, we now define the advantage of the adversary for all $T \subset [n]$ and $|T| \leq t$ in all valid experiments above as

$$Adv_{\mathcal{A}}^{t-Sec}(\mathcal{PVFSS}, F, n, \lambda) = \left|\Pr[\text{EXP}_{\mathcal{A}}^{t-Sec}[\mathcal{PVFSS}, F, n, \lambda] = 1] - 1/2\right|$$

We say that $\mathcal{PVFSS}$ is a t-secure $\mathcal{PVFSS}$ scheme if $Adv_{\mathcal{A}}^{Sec}(\mathcal{PVFSS}, F, n, \lambda) \leq neg(\lambda)$

Definition 7. ($\mathcal{PVFSS}$ Strong Unforgeability). *Let $\mathcal{PVFSS}$ be a verifiable function secret sharing scheme for a function $f \in F$ among n parties, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ be a triple-tuple of probabilistic polynomial-time machines. We define security via the following experiment.*

$$
\begin{aligned}
&\textit{Experiment } \mathbf{EXP}_{\mathcal{A}}^{Verify}[\mathcal{PVFSS}, F, n, \lambda]: \\
&(SK_f, VK_f) \leftarrow \text{Setup}(1^\lambda, f); \\
&(EK_1, \cdots, EK_n) \leftarrow \text{keyGen}(f, n, SK_f); \\
&For\ i = 1, \cdots \\
&\quad x_i \leftarrow \mathcal{A}_0(EK_1, \cdots, EK_n, x_1, \omega_{f(x_1)}, \cdots, x_{i-1}, \omega_{f(x_{i-1})}); \\
&x^* \leftarrow \mathcal{A}_1(EK_1, \cdots, EK_n, x_1, \omega_{f(x_1)}, \cdots); \\
&\hat{\omega}_{f(x^*)} \leftarrow \mathcal{A}_2(EK_1, \cdots, EK_n, x^*, \omega_{f(x^*)}, \cdots, x_1, \omega_{f(x_1)}, \cdots); \\
&\hat{b} \leftarrow \text{Verify}(\hat{\omega}_{f(x^*)}, y, VK_f, x^*); \\
&If\ \hat{\omega}_{f(x^*)} \neq \omega_{f(x^*)}\ and\ \hat{b} = 1,\ output\ '1',\ else\ '0';
\end{aligned}
$$

The adversary succeeds if it produces an output that convinces the verification algorithm to accept on the wrong witness for a given input value. We now define the advantage of the adversary in the experiment above as:

$$Adv_{\mathcal{A}}^{Verify}(\mathcal{PVFSS}, f, n, \lambda) = \Pr\left[Exp_{\mathcal{A}}^{Verify}\left[\mathcal{PVFSS}, f, n, \lambda\right] = 1\right]$$

We say that $\mathcal{PVFSS}$ is strong unforgeable f, if $Adv_{\mathcal{A}}^{Verify}(\mathcal{PVFSS}, f, n, \lambda) \leq neg(\lambda)$.

Likewise, public verifiable function secret sharing is private when the outputs of the problem generation algorithm over two different inputs are indistinguishable, i.e., an adversary cannot decide which encoding is the correct one for a given input. More formally:

Definition 8. ($\mathcal{PVFSS}$ I/O Privacy). *Let $\mathcal{PVFSS}$ be a public verifiable function secret sharing scheme for a function $f \in F$ among n parties, and let $\mathcal{A}(\cdot) = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ be a triple-tuple of probabilistic polynomial-time machines. We define privacy via the following experiment.*

$$
\begin{aligned}
&\textit{Experiment } \mathbf{EXP}_{\mathcal{A}}^{Priv}[\mathcal{PVFSS}, F, n, \lambda] : \\
&(SK_f, VK_f) \leftarrow \mathrm{Setup}(1^\lambda, f); \\
&(EK_1, \cdots, EK_n) \leftarrow \mathrm{keyGen}(f, n, SK_f); \\
&\qquad (x_0, x_1, \cdots) \leftarrow \mathcal{A}_0(1^\lambda, f); \\
&\qquad index \leftarrow \mathcal{A}_1(1, \cdots, n); \\
&b \in_R \{0, 1\}; \\
&y_{index,b} \leftarrow \mathrm{Eval}(EK_{index}, x_b); \\
&\qquad \hat{b} \leftarrow \mathcal{A}_2(f, n, x_0, x_1, y_{index,b}); \\
&If\ b = \hat{b},\ output\ '1',\ else\ '0'
\end{aligned}
$$

Based on a typical indistinguishability argument, we now define the advantage of the adversary in the experiment above as:

$$Adv_{\mathcal{A}}^{Priv}(\mathcal{PVFSS}, F, n, \lambda) = \left|\Pr[\mathrm{EXP}_{\mathcal{A}}^{Priv}[\mathcal{PVFSS}, F, n, \lambda] = 1] - 1/2\right|$$

We say that $\mathcal{PVFSS}$ is a private $\mathcal{PVFSS}$ scheme if $Adv_{\mathcal{A}}^{Priv}(\mathcal{PVFSS}, F, n, \lambda) \leq neg(\lambda)$.

A similar definition can be made for output privacy.

4 Construction for Point Function

We begin in section 4.1 by showing a main construction of a public verifiable function secret sharing scheme for point functions. While section 4.2 make a correctness and security proof required by our protocol. Section 4.3 presents a performance analysis, including key size and algorithm efficiency. In algorithm analysis stage, a few comparative experiments are conducted on each algorithm between FSS and $\mathcal{PVFSS}$. Finally, the resluts manifest that our scheme possesses lots of advantages.

4.1 Main Construction

[10] introduces a n-parties function secret sharing(FSS) scheme for point function. This scheme allows a dealer to split an arbitrary point function $P_{x,y}(\hat{x})$ over

an Abelian group $\mathbb{G}$ into n sub-functions $f_i : \{0,1\}^* \to \mathbb{G}$, $1 \leq i \leq n$, such that $\sum_{i=1}^{n} f_i(x) = P_{x,y}(\hat{x})$. Pairs of parties' output shares for each input x can be simulated given just the corresponding outputs $P_{x,y}(\hat{x})$, and thus each sub-function provides no information about the secret function individually. This scheme assumed that the servers are absolutely *reliable*. However, there is always some deviations in reality that severs may misbehave or return a malicious result of sub-function. As a result, the client will reconstruct a wrong secret. Therefore, in what ways can these issues be solved? [25,26,27,28] put forward a significant idea that attach some luxurious information as a witness to check the correctness of computation. More preciously, in [25], an additional function $r(x)$ was added to an overall witness $\omega(x) = af(x) + r(x)$ by algebraic operation, where the secret keys are a and $r(x)$. Noted that, anyone will get zero-knowledge of secret keys from the integrated witness $\omega(x)$ except the owner of secret keys.

As an initial attempt towards solution, our scheme makes an analogy with [25]. That is to say, it aims at constructing a witness in the shape of $ay + r(x)$, where secret keys a and $r(x)$ are kept alone by the dealer. Nevertheless, FSS scheme is on the basis of the Pseudo-Random Generation (PRG) $H : \{0,1\}^n \to \{0,1\}^m, n, m \in Z^+$ over an Abelian Group $\mathbb{G}$. The PRG [29,30] can be considered as a hash function [31,32]. Hence, the overall witness $H_\omega(x)$ is equal to the form of $aH_y(x) + H_r(x)$. For the sake of convenience, let H represent $H(x)$. Perhaps frustrating, one difficulty is that due to the unpredictability of PRG's output, whether two completely different hash functions H_y and H_r can map to one single and unbroken hash function H_ω by algebraic operation. If not, secret will be exposed to the public. The main reason is that the witness, $aH_y + H_r$, can't be able to combined into a whole. Another difficulty is that how to find and construct H_ω, if there exists.

In order to handle the mentioned issues, we firstly prove theoretically the existence of entirety PRG H_ω, which is made up of a, H_r and H_y, where H_y and H_r are represented to some PRGs over Abelian Group $\mathbb{G}_y$ abd $\mathbb{G}_r$ respectively.

Proof. By algebraic operation among H_y and H_r, the mapped group $\mathbb{G}_\omega$ is also consider as an Abelian Group, and the identity element e_ω is equal to $ae_y + e_r$. □

Now, the main task is to construct an overall PRG H_ω such that $H_\omega = aH_y + H_r$ without revealing the secret keys. PRG can be consider as a hash function, which can map data of arbitrary size to data of fixed size. Furthermore, whatever mechanism we don't have to know, we just only invoke it. Therefore, it is difficult to find this approach directly or brutally. For constructing an overall witness, the dealer should not have to understand the concrete implementation methods or mechanisms about two different hash algorithm. Another reason is that constructing H_ω by learning two different mechanisms may cost substantially much more effort than function secret sharing scheme. In short, this method is unreal with overwhelming probability.

One possible approach to correct this deficiency is shown as follows. In the natural way, change witness PRG $H_\omega = aH_y + H_r$ into a witness function

$P_{x,ay+r}(\hat{x})$. When it comes to this innovation, one common method, the dealer will split and distribute two point functions $P_{x,y}(\hat{x})$ and $P_{x,ay+r}(\hat{x})$, may occur to us and is used to verify the correctness of the computation immediately. At a high level, $ay + r$ can be considered as a function concerning y (i.e. $f(y)$) in a much bolder way. Besides, there is no doubt that this construction may cost twice effort than just only one function.By upgrading and extending the idea, we construct an efficiency improved scheme, and whose efficiency is asymptotic to that of FSS. A detailed description is given as below.

Algorithm $(SK_f, VK_f) \leftarrow \text{Setup}(1^\lambda, f)$: Suppose that the function is a point function $P_{x,y}$. The dealer, on input the point function $P_{x,y}$ with a security parameter, invokes a public hash algorithm $h_{VK_f}(\cdot)$, which take as input y, to generate a random value $h_{VK_f}(y)$ (of λ bits) as a secret key SK_f and a public verifiable key VK_f.

Algorithm $(EK_1, \cdots, EK_n) \leftarrow \text{KeyGen}(f, n, SK_f)$: The dealer, on input the target function , piece numbers n and a secret key $SK_f = r$, run KeyGen algorithm once to output n evaluation $(EK_1, \cdots, EK_n)$ keys $P_{x,y}$, which are respectively represented sub-functions $(f_1, \cdots, f_n)$. We describe a construction which is an upgraded of FSS's in Algorithm 1.

Algorithm 1 $\text{KeyGen}(f, n, SK_f)$

1: Let $G : \{0, \cdots, p-1\}^\lambda \rightarrow \{0, \cdots, p-1\}^{|y| \cdot p^\mu}$ be a PRG (μ is defined in line 2).

2: Let $\mu \leftarrow \lceil \frac{1}{2} \log(2^{|x|} \cdot 2^{n-1}) \rceil$ and $\nu \leftarrow |x| - \mu$.

3: Parse x as a pair $x = (\alpha, \beta)$, $\alpha \in [p^\mu]$, $\beta \in [p^\nu]$.

4: Choose p^μ arrays $A_1, \cdots, A_{p^\mu}$ randomly, s.t. $A_\alpha \in_R O_n$ and $A_{\hat{\alpha}} \in_R E_n$ for $\hat{\alpha} \neq \alpha$.

5: Choose randomly and independently $p^\mu \cdot p^{n-1}$ seeds $s_{1,1}, \cdots, s_{1,p^{n-1}}, \cdots, s_{p^\mu,1}, \cdots, s_{p^\mu,p^{n-1}} \in \{0, \cdots, p-1\}^\lambda$.

6: Choose p^{n-1} random strings $cw^0_1, \cdots, cw^0_{p^{n-1}}, cw^1_1, \cdots, cw^1_{p^{n-1}} \in \{0, \cdots, p-1\}^{|y| \cdot 2^\mu}$ s.t. $\sum_{j=1}^{p^{n-1}} cw^0_j + G(s_{\alpha,j}) = e_\beta \cdot y \mod p^{|y| \cdot p^\mu}$ and $\sum_{j=1}^{p^{n-1}} cw^1_j + G(s_{\alpha,j}) = e_\beta \cdot SK_f \mod p^{|y| \cdot p^\mu}$.

7: Set $\sigma_{i,\hat{\alpha}} \leftarrow (s_{\hat{\alpha},1} \cdot \mathrm{A}_{\hat{\alpha}}[\mathrm{i}, 1]) || \cdots || (s_{\hat{\alpha},p^{p-1}} \cdot \mathrm{A}_{\hat{\alpha}}[\mathrm{i}, \mathrm{p}^{n-1}])$ for $1 \leq i \leq n$, $1 \leq \hat{\alpha} \leq p^\mu$.

8: Set $\sigma_i = \sigma_{i,1} || \cdots || \sigma_{i,p^\mu}$ for $1 \leq i \leq n$.

9: $EK_i = (\sigma_i, cw^0_1, \cdots, cw^0_{p^{n-1}}, cw^1_1, \cdots, cw^1_{p^{n-1}})$ for $1 \leq i \leq n$.

10: Return $(EK_1, \cdots, EK_n)$.

Algorithm $(f_i(x), \omega_{f_i(x)}) \leftarrow \text{Eval}(i, EK_i, x)$: The server, on receiving index i, a evaluation key EK_i and input string x, returns a value $f_i(x)$ together with a witness $\omega_{f_i(x)}$ both at input x. For the sake of convenience and intuitive expression, we directly achieve the detailed implement in Algorithm 2.

Algorithm 2 $\text{Eval}(i, EK_i, x)$

1: Let $G : \{0, \cdots, p-1\}^{\lambda} \rightarrow \{0, \cdots, p-1\}^{|y| \cdot p^{\mu}}$ be a PRG (μ is defined in line 2).

2: Let $\mu \leftarrow \lceil \frac{1}{2} \log(2^{|x|} \cdot 2^{n-1}) \rceil$ and $\nu \leftarrow |x| - \mu$.

3: Parse x as a pair $x = (\hat{\alpha}, \hat{\beta})$, $\hat{\alpha} \in [p^{\mu}]$, $\hat{\beta} \in [p^{\nu}]$.

4: Regard EK_i as $EK_i = (\sigma_i, cw_1^0, \cdots, cw_{p^{n-1}}^0, cw_1^1, \cdots, cw_{p^{n-1}}^1)$.

5: Parse σ_i as $\sigma_i = s_{1,1} || \cdots || s_{1,p^{n-1}} || \cdots || s_{p^{\mu},1} \cdots || s_{p^{\mu},p^{n-1}}$.

6: Let $y_i \leftarrow \sum_{j=1, S_{\alpha,j} \neq 0}^{p^{n-1}} cw_j^0 + G(s_{\alpha,j}) \bmod p^{|y| \cdot p^{\mu}}$ and $\omega_{f_j(x)} \leftarrow \sum_{j=1, S_{\alpha,j} \neq 0}^{p^{n-1}} cw_j^1 + G(s_{\alpha,j}) \bmod p^{|y| \cdot p^{\mu}}$.

7: Return $(y_i[\hat{\beta}], \omega_{f_i(x)}, [\hat{\beta}])$.

Algorithm $(y, \omega_{f(x)}) \leftarrow \text{Dec}(y_1, \omega_{f_1(x)}, \cdots, y_n, \omega_{f_n(x)})$: The client, on receiving n results pair $(y_i, \omega_{f_i(x)})$, compute the evaluation y, where $y = f(x)$ is the output of the function y at input x, together with a witness $\omega_{f(x)}$ at point x.

Algorithm $1 or \perp \leftarrow \text{Verify}(\omega_{f(x)}, y, VK_f, x)$: The client, on receiving the witness $\omega_{f(x)}$, the result y and the public verifiable key VK_f outputs

$$y = \begin{cases} 1 & if\ \omega_{f(x)} = H_{VK_f}(y) \\ \perp & if\ \omega_{f(x)} \neq H_{VK_f}(y) \end{cases}$$

4.2 Correctness and Security Proof

Proof of correctness: Algorithm $\text{KeyGen}(f, n, SK_f)$ generates $(EK_1, \cdots, EK_n)$, then $\text{Dec}(\hat{y}_1, \cdots, \hat{y}_n) = \sum_{i=1}^{n} \text{Eval}(i, EK_i, x)[0] = \sum_{i=1}^{n} (\sum_{j=1, S_{\alpha,j} \neq 0}^{p^{n-1}} cw_j^0 + G(s_{\alpha,j}))[0, \hat{\beta}] = \sum_{i=1}^{n} y_i[\hat{\beta}] \bmod p^{|y| p^{\mu}}$ is same as $\text{Dec}(\hat{\omega}_{f_1(x)}, \cdots, \hat{\omega}_{f_n(x)})$. For each $\hat{x} = (\hat{\alpha}, \hat{\beta})$,

Case 1 $\hat{\alpha} \neq \alpha$: Since $A_{\hat{\alpha}}$ belongs to E_p, each of the terms $cw_j^0 + G(s_{\alpha,j})$ appears an even number of times such that all of them will cancel out (i.e. $\text{Dec}(\hat{y}_1, \cdots, \hat{y}_n) = 0$).

Case 2 $\hat{\alpha} = \alpha$: Due to the definition of correction elements, we have

Case 2.1 $\hat{\beta} \neq \beta$ such that $\hat{y} = \text{Dec}(\hat{y}_1, \cdots, \hat{y}_n) = 0$, otherwise, we have

Case 2.2 $\hat{\beta} = \beta$ such that $\hat{y} = \text{Dec}(\hat{y}_1, \cdots, \hat{y}_n) = y$
Hence, all of these cases satisfy $h_{VK_f}(\hat{y}) = \omega_{f(x)}$. □

Proof of t-Security: Generally speaking, $\mathcal{PVFSS}$ is t-security, because all evaluation keys $(EK_1, \cdots, EK_n)$ are completely independent and random (i.e. pseudo random). Or rather, each strict subset of evaluation includes at most n-1 strings $\sigma_i = \sigma_{i,1}|| \cdots ||\sigma_{i,p^\mu}$. The distribution of seeds reflects the distribution of 1 bits in the *i-th* row of $A_{\hat{\alpha}}$, in string $\sigma_{i,\hat{\alpha}}$, which is uniformly distributed. Meanwhile, all of the $p^\mu \cdot p^{n-1}$ seeds are sampled randomly and distributed identically. Therefore, the views of the string σ_i do not give any knowledge about α. Together, these elements $cw_j^0 + G(s_{\alpha,j})$ satisfy $\sum_{i=1}^{n} (\sum_{j=1, S_{\alpha,j} \neq 0}^{p^{n-1}} cw_j^0 + G(s_{\alpha,j}))[0, \beta] = y$. However, one of evaluation keys $(EK_1, \cdots, EK_n)$ does not include at least one seed. Since total $2(n-1) \cdot p^{n-1}$ random strings $cw_j^0, G(s_{\alpha,j})$ are indistinguishable, the task, finding the appropriate path, isn't as easy as it seems regardless of all the correction factors together. □

Proof of Strong Unforgeability:

Initialization Stage: The challenger runs Setup$(1^\lambda, f)$ algorithm, and distributes the public verifiable key VK_f to the adversary $\mathcal{A}$. Algorithm KeyGen(f, n, SK_f) generates evaluation keys $(EK_1, \cdots, EK_n)$ and send them to the adversary $\mathcal{A}$.
Simulator Stage: Before adversary $\mathcal{A}$ first queries the evaluation algorithm Eval(i, EK_i, x), the simulator $S = (S_{input}, S_1, \cdots, S_n, S_{Dec})$ performs the following for polynomial-many times.

i. The simulator S_{input} needs to choose a random point x.

ii. On receiving input x, the simulator S_i evaluates the *i-th* sub-function at x, and create the corresponding witness $\omega_{f_i(x)}$ from $i = 1$ to n.

iii. After this, the simulator S_{Dec} will execute the decoder algorithm Dec(y_1, $\omega_{f_1(x)}, \cdots, y_n, \omega_{f_n(x)}$) and return the result of secret function $f(x)$ and entire witness $\omega_{f(x)}$ back to the adversary $\mathcal{A}$.
Challenge Stage: The adversary $\mathcal{A}$ picks a input point x^* The simulator S executes the same processes as stated Stage 1 above. At last, the whole witness $\omega_{f(x^*)}$ and the result $f(x^*)$ is public to the adversary $\mathcal{A}$. At the same time, the adversary $\mathcal{A}$ outputs a forgery $\hat{\omega}_{f(x^*)}$) for the witness at point x^* in order to pass the verification algorithm.

Supposing the forgery is successful, which means $1 \leftarrow \text{Verify}(\hat{\omega}_{f(x^*)}, y^*, VK_f, x^*)$ (i.e. $H_{VK_f}(f(x^*)) = \hat{\omega}_{f(x^*)}, 1 \leftarrow \text{Verify}(\omega_{f(x^*)}, y^*, VK_f, x^*)$), (i.e. $H_{VK_f}(f(x^*)) = \omega_{f(x^*)}$) and $\omega_{f(x^*)} \neq \hat{\omega}_{f(x^*)}$. Therefore, the following must be true:

$$H_{VK_f}(f(x^*)) \neq H_{VK_f}(f(x^*)).$$

Noted that it breaks the second pre-image resistance property for the hash function. □

Proof of I/O Privacy: The adversary $\mathcal{A}$ picks two input point x_0, x_1, each simulator S_i performs evaluation algorithm $\text{Eval}(i, EK_i, x)$ twice, for each input x_0, x_1, to outputs shares $y_{i,0}, y_{i,1}$, respectively. The shares $y_{i,0}, y_{i,1}$ are random strings, which are simulated given just the corresponding outputs $f(x)$. One step further, the results of sub-function are indistinguishable.

A similar proof can be made for output privacy. □

4.3 Performance Analysis

If we choose two point functions and perform FSS scheme, respectively. There is no doubt that all performance, including key length and algorithm efficiency, must cost twice effort than just only one. We introduce a primitive called Public Verifiability Function Secret Sharing Scheme with asymptotic efficiency of FSS's. The scheme analysis will be presented in the following two aspects: key length analysis and algorithm analysis.

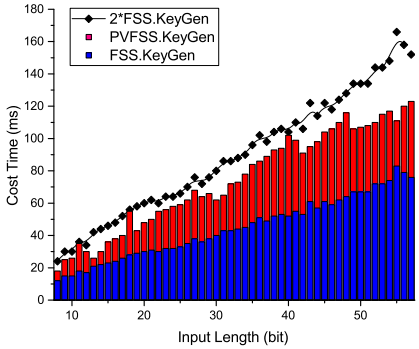
Key Length Analysis: Note that the length of a key EK_i in [10] that KeyGen outputs is a sum of the length of δ_i, which is $\lambda \cdot p^{\mu} \cdot p^{n-1}$ and the length of the correction factor is $|y| \cdot p^{\mu} \cdot p^{n-1}$. Therefore, the key size is $O(p^{\mu} \cdot p^{n-1}(\lambda \cdot + |y|))$. In our $\mathcal{PVFSS}$ scheme, the length of a key EK_i is similar to that in FSS. Nonetheless, we point out a large difference that our correction factor is twice of FSS in length. Thus, the key size is $O(p^{\mu} \cdot p^{n-1}(\lambda \cdot + 2\,|y|)) \ll O(2 \cdot p^{\mu} \cdot p^{n-1}(\lambda \cdot + |y|))$.

Algorithm Analysis: We analysis algorithms of $\mathcal{PVFSS}$ scheme making lots of comparative experiments. The two major metrics that we use to evaluate algorithms are input length and output length. Finally, the results of the experiments manifest or proof two significant benefits of $\mathcal{PVFSS}$. One is that the $\mathcal{PVFSS}$ efficiency is asymptotic to the FSS's. Another is that verifying the result, which is constant in time, costs substantially less time than performing the computation locally or by himself. We conduct all experiment on all servers with Intel(R) Core(TM)i7-2600 CPU @ 3.40GHz, 8.00 GB RAM and a computationally weak client with Intel(R) Core(TM)2 Duo CPU P9300 2.27GHz, 2GB RAM. Fig. 1(a), 1(b), 1(c), 1(d), contrast of $\mathcal{PVFSS}$ and FSS, plot the cost time influenced by input length on algorithm KeyGen, algorithm Eval, algorithm Dec & Verify and total time, respectively. In the above comparisons, each algorithm efficiency is closed to the FSS's in our scheme $\mathcal{PVFSS}$. Intuitively, Fig. 2 shows our scheme is not affected by the output length, which is similar to the behavior of FSS. A contrast of between algorithm Verify and algorithm Eval on cost time is made in Fig. 3. Also, the efficiency of algorithm Verify is constant, which is far less than that of algorithm Eval. In short, our scheme can be applied into practice such as cloud computing and outsource computing.

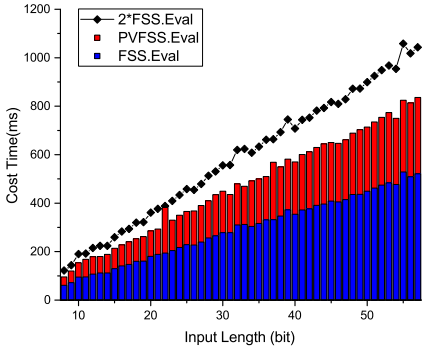
5 Applications

5.1 Cloud Computing

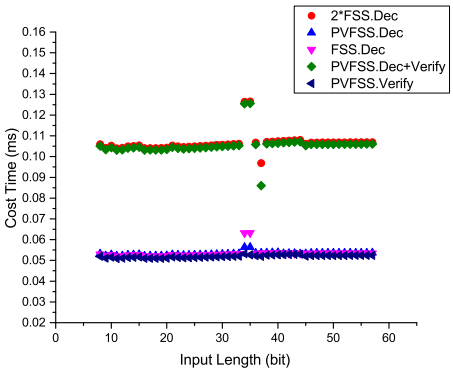
In the scenario of cloud computing a computationally weak client requests a semi-trusted or untrusted but more powerful cloud to perform some complex tasks.



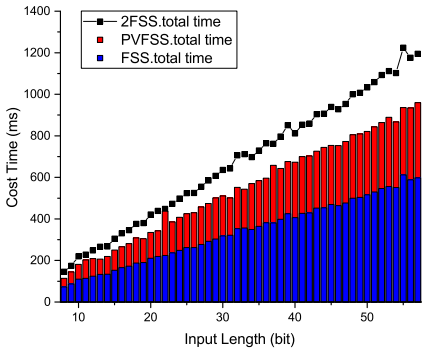
(a) Algorithm KeyGen Cost Time



(b) Algorithm Eval Cost Time



(c) Algorithm Dec & Verify Cost Time



(d) Total Time

Fig. 1. $\mathcal{PVFSS}$ $\mathcal{V}$s FSS on Input Length

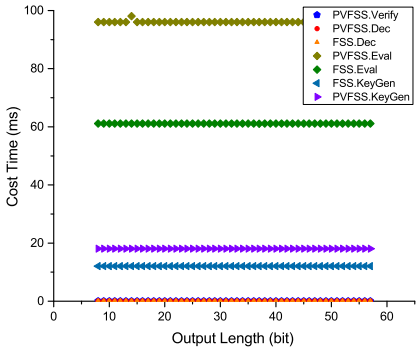


Fig. 2. $\mathcal{PVFSS}$ $\mathcal{V}$s FSS on Output Length

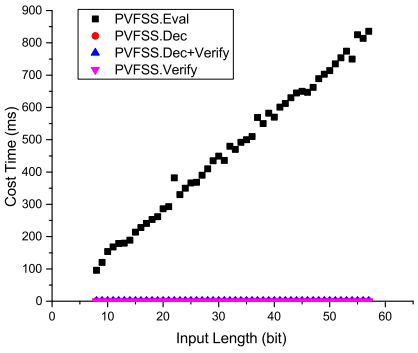


Fig. 3. Eval $\mathcal{V}$s Verify in $\mathcal{PVFSS}$

The latter is expected to return the results. This paradigm is in the amortized model of [12], in which the client invests a one-time expensive computation so that the computational payment will be cost and shared by the server on many inputs. Cloud computing relies on sharing powerful resources rather than having local or personal devices to handle expensive applications, which bring so many advantages and benefits., Without it, for instance, we should require buying hundreds of physical devices when we process some expensive computation and then sell them back as long as we do not need. Similarly, whatever physical devices you possess, all of these low or weak devices can process heavy work. such as mobilephones and wearable smart devices. Our $\mathcal{PVFSS}$ scheme can be applied to the cloud computing. As an analogy between $\mathcal{PVFSS}$ and cloud computing where this might come into handy, the dealer is similar to the delegator and key generation stage can be considered as pre-processing stage (amortized model) in cloud computing. Each piece represents an onerous task, which is outsourced to a different powerful cloud server. Then the cloud responds with the evaluation of the sub-function together with a witness. Finally, the client checks that the result provided by the server is indeed the result of the function computed on the input provided. It is worthy to note that constant verifiable time is absolutely less than the that of performing the sub-function locally or by himself.

5.2 Other Applictations

There exist plenty of scenarios being applied to our scheme. For example, We can extend a E-Voting to public verifiable E-Voting so that the servers can cheat the voting institution. The institution initial a election function, which is a point function. The input of this function is a candidate's id, and output '1'. After the voter polls for the candidate, the counting agent can check and count the bills. Similarly, [23] introduced three applications: computationally private information retrieval, private information retrieval by keywords and worst-case to average-case reduction. As well as, all of these not only are suitable for our scheme, but also are added three properties above.

6 Conclusions and Future Work

In this paper, a novel primitive called public verifiable function secret sharing ($\mathcal{PVFSS}$) was introduced, which aimed to guarantee the correctness and integrity of the computation outsourced to an untrusted server. $\mathcal{PVFSS}$ allows a dealer to split a secret function into some pieces, and then distribute and outsource them to different servers. On issuing a query, the server returns a result together with a witness to the client. The client can use the proof to verify the correctness of the result. Meanwhile, we proposed three important properties as following: high efficiency, public delegation and public verification. The primitive adopted a point function to construct our scheme, which captures these three new properties. This new construction for point function held a good performance, especially in the verification cost. The cost time of verification is constant, and

not depending on the input length. Moreover, a few comparative experiments are conducted on each algorithm between FSS and $\mathcal{PVFSS}$. The comparisons were done using different algorithms on different input samples. Nevertheless, the results of the experiments manifest the two significant benefits of $\mathcal{PVFSS}$. One that is the $\mathcal{PVFSS}$'s efficiency is asymptotic to that of FSS. Another is that verifying the result, which is constant in time, costs substantially less time than performing the computation locally. In short, $\mathcal{PVFSS}$ is indeed practical. However, some interesting and challenging open problems still stems from this work. We will further adopt several functions, such as univariate polynomial and multivariate polynomial, and construct fine-grained methods so that which server returns a malicious result could be found.

References

1. Shamir, A.: How to share a secret. Communications of the ACM. 22(11), 612–613 (1979)
2. Mignotte, M.: How to share a secret. Cryptography. 371–375 (1983)
3. Chien, H. Y. and Jinn-Ke, J.A.N. and Tseng, Y. M.: A practical (t, n) multi-secret sharing scheme. IEICE transactions on fundamentals of electronics, communications and computer sciences. 83(12), 2762–2765 (2000)
4. Pang, L. J. and Wang, Y. M.: A new (t, n) multi-secret sharing scheme based on Shamir's secret sharing. Applied Mathematics and Computation. 167(2), 840–848 (2005)
5. Armbrust, M. and Fox, A. and Griffith, R. and Joseph, A. D. and Katz, R. and Konwinski, A. and Lee, G. and Patterson, D. and Rabkin, A. and Stoica, I. and others: A view of cloud computing. Communications of the ACM. 53(4), 50–58 (2010)
6. Buyya, R. and Yeo, C. S. and Venugopal, S.: Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on. 5–13 (2008)
7. Mell, P. and Grance, T.: The NIST definition of cloud computing. (2011)
8. El-Sayed, S. M. and Kader, H. M. A. and Hadhoud, M. M. and AbdElminaam, D. S.: Mobile Cloud Computing Framework for Elastic Partitioned/Modularized Applications Mobility.
9. Boyle, E. and Gilboa, N. and Ishai, Y.: Function Secret Sharing. Advances in Cryptology-EUROCRYPT 2015. 337–367 (2015)
10. Chor, B. and Goldwasser, S. and Micali, S. and Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults. 6th Annual Symposium on Foundations of Computer Science (sfcs 1985). (1985)
11. Gennaro, R. and Gentry, C. and Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. Advances in Cryptology–CRYPTO 2010. 465–482 (2010)
12. Parno, B. and Raykova, M. and Vaikuntanathan, V.: How to delegate and verify in public: Verifiable computation from attribute-based encryption. Theory of Cryptography. 422–439 (2012)
13. Parno, B. and Howell, J. and Gentry, C. and Raykova, M.: Pinocchio: Nearly practical verifiable computation. Security and Privacy (SP), 2013 IEEE Symposium on. 238–252 (2013)

14. Papamanthou, C. and Shi, E. and Tamassia, R.: Signatures of Correct Computation. Theory of Cryptography. 2, 1–33 (2013)
15. Papamanthou, C. and Shi, E. and Tamassia, R.: Signatures of correct computation. Theory of Cryptography. 2, 1–33 (2013)
16. Desmedt, Y. G.: Threshold cryptography. European Transactions on Telecommunications. 5(4), 449–458 (1994)
17. Blakley, G. R. and others: Safeguarding cryptographic keys. Proceedings of the national computer conference. 48, 313–317 (1979)
18. Asmuth, C. and Bloom, J.: A modular approach to key safeguarding. IEEE transactions on information theory. 29(2), 208–210 (1983)
19. Stadler, M.: Publicly verifiable secret sharing. Advances in CryptologyEUROCRYPT'96. 190–199 (1996)
20. Laud, P. and Kamm, L.: Practical Applications of Secure Multiparty Computation. Applications of Secure Multiparty Computation. 13,(246) - (2015)
21. Goldreich, O.: Secure multi-party computation. Manuscript. Preliminary version. (1998)
22. Smart, N. P.: Secure Multi-party Computation. Cryptography Made Simple. 439–450 (2016)
23. Nair, D. G. and Binu, V. P. and Kumar, G. S.: An Effective Private Data storage and Retrieval System using Secret sharing scheme based on Secure Multi-party Computation. arXiv preprint arXiv:1502.07994. (2015)
24. Tzeng, W. G.: A secure fault-tolerant conference-key agreement protocol. Computers, IEEE Transactions on. 51(4), 373–379 (2002)
25. Kim, Y. and Perrig, A. and Tsudik, G.: Simple and fault-tolerant key agreement for dynamic collaborative groups. Proceedings of the 7th ACM conference on Computer and communications security. 235–244 (2000)
26. Tzeng, W. G.: A practical and secure fault-tolerant conference-key agreement protocol. Public Key Cryptography. 1–13 (2000)
27. Yi, X.: Identity-based fault-tolerant conference key agreement. Dependable and Secure Computing, IEEE Transactions on. 1(3), 170–178 (2004)
28. Tseng, Y. M.: A communication-efficient and fault-tolerant conference-key agreement protocol with forward secrecy. Journal of Systems and Software. 80(7), 1091–1101 (2007)
29. Gilboa, N. and Ishai, Y.: Distributed point functions and their applications. Advances in Cryptology–EUROCRYPT 2014. 640–658 ()
30. Fiore, D. and Gennaro, R.: Publicly verifiable delegation of large polynomials and matrix computations, with applications. Proceedings of the 2012 ACM conference on Computer and communications security. 501–512 (2012)
31. Applebaum, B. and Ishai, Y. and Kushilevitz, E.: From secrecy to soundness: Efficient verification via secure computation. Automata, languages and Programming. 152–163 (2010)
32. Benabbas, S. and Gennaro, R. and Vahlis, Y.: Verifiable delegation of computation over large datasets. Advances in Cryptology–CRYPTO 2011. 111–131 (2011)
33. Gennaro, R. and Gentry, C. and Parno, B.: Non-interactive verifiable computation: Outsourcing computation to untrusted workers. In Proceedings of CRYPTO. (2010)
34. Impagliazzo, R. and Levin, L. A and Luby, M.: Pseudo-random generation from one-way functions. Proceedings of the twenty-first annual ACM symposium on Theory of computing. 12–24 (1989)
35. Blum, L. and Blum, M. and Shub, M.: A simple unpredictable pseudo-random number generator. SIAM Journal on computing. 15(2), 364–383 (1986)

36. Coron, J.S. and Dodis, Y. and Malinaud, C. and Puniya, P.: Merkle-Damgård revisited: How to construct a hash function. Advances in Cryptology–CRYPTO 2005. 430–448 (2005)
37. Black, J. and Rogaway, P. and Shrimpton, T.: Black-box analysis of the block-cipher-based hash-function constructions from PGV. Advances in Cryptology-CRYPTO 2002. 320–335 (2002)
38. Papamanthou, C. and Shi, E. and Tamassia, R.: Signatures of Correct Computation. Theory of Cryptography. Springer Berlin Heidelberg 2013. 222–242 (2013)