Authenticated Secret Sharing with Penalty, and Application to Secure Cloud Computing

Abstract. Secret sharing based multiparty computation (MPC) has been regarded as a promising approach for countering the risks in cloud computing. The client can split its data into shares, store the shares with different clouds, and later instruct the clouds to compute securely on the shares. In this paper, we study the problem that when *all* the clouds have been compromised by an external adversary, how to enable automatic penalty such that the client can be compensated for the loss. We propose *Authenticated Secret Sharing with Penalty* (ASSP). The core idea of ASSP is to pack the client's data and a cryptocurrency secret key from the clouds together in the shares. In consequence, if the client's data is leaked, so does the servers' key. By utilizing a cryptocurrency, the leaked keys can be translated into financial penalties easily. Besides, in the case of at least one server is honest, ASSP still enables share-based secure computation without risking the servers' cryptocurrency. This is achieved by incorporating a novel share authentication mechanism so that the client can ensure the integrity of the outsourced data and the correctness of the outsourced computation. To our best knowledge, ASSP is the first secret sharing scheme that *simultaneously* achieves prior two seemingly incompatible properties. We also report on the performance of ASSP based on a prototype implementation.

Keywords: authenticated secret sharing · cryptocurrency · outsourced computation · financial penalty

1 Introduction

Secret sharing based Multiparty Computation (MPC) has been widely regarded as a promising tool for solving many real-world security and privacy problems [5]. A problem at the top of the list is secure cloud computing. The risks of storing and processing data in untrusted clouds have become a major concern of the cloud users. With secret sharing based MPC, the client can split its data into shares, store each share with a different cloud, then instruct the clouds to run an MPC protocol to compute jointly using the shares as input. As long as the data is kept in the shared form, the client can be assured about the privacy of its data. With a share authentication mechanism, the client can also verify the integrity of the stored data and computation results.

While all sounds good, one should keep in mind that *secret sharing is not encryption.* If data is encrypted, as long as the data owner can keep the key secure, the data remains secure even if an adversary can obtain the ciphertext. But with secret sharing, the adversary can immediately recover the plaintext once gathered enough shares, and the data owner can do nothing about it. This is why when using secret sharing to secure data in the cloud, the client must find multiple *non-colluding* cloud providers to ensure none of them can get enough shares to recover the data. In reality, the non-colluding assumption is often justifiable as the cloud providers are usually competitors and will not risk their business to cooperate with others in such an obvious illegal activity, i.e. stealing the client's data. It might even be possible to intensify the distrust among the clouds with economic incentives, to make collusion even harder [18].

However, the non-colluding assumption does not preclude attacks by an *external adversary* who tries to compromise the data stored in the clouds. Although compromising one cloud does not immediately lead to the breach of the client's data when the data is shared, eventually the data will be leaked once the adversary has compromised enough clouds and gathered enough shares. This is not paranoid: a recently disclosed breach of several large cloud providers including HPE and IBM by the same hacker group proved that the threat is real [9]. The client can do little about it because the control is off its hands.

While the client loses direct control over the security of its data, it can still exert some indirect control through penalties. At present, certain liabilities and penalties can be stated in contracts or Service Level Agreements [43], so that the cloud will be punished for their negligence and the client can be compensated financially. However, how to enforce the penalties is a challenge [41]. Unlike events such as service disruptions which are obviously noticeable, data security breaches are often not. Moreover, enforcing the penalties through legal actions is expensive and time consuming. It is also difficult to establish the responsibility because it is often the cloud providers who control the information about the data breach. The lack of accountability and the difficulty in enforcement lead to a pressing need for technical mechanisms that can enforce penalties in the event of a data breach automatically.

High-level idea Our goal is to incorporate automatic penalty, in the presence of an external adversary, into secret sharing based MPC for secure cloud computing. To achieve this, we pack the client's data with a cryptocurrency secret key that is jointly generated by the clouds, and embed the bundle in (MPC-compatible) secret shares, so that if an external adversary can recover the client's data from the shares, it can also recover the clouds' secret key. The adversary can then use the clouds' key to spend some coins pre-deposited by the clouds, in a way that a portion of the coins will go to the client (as the client's compensation) and the rest will go to the adversary (to incentivize the adversary to transfer the coins).

Challenges How to enable automatic penalty while still allow computing on client's data by the clouds was the main challenge we encountered. For example, a naive solution is to use threshold cryptography such that the clouds jointly generate a public/secret encryption key pair, then use the public key to encrypt the cryptocurrency secret key and publish the ciphertext. Then the client encrypts its data under this public key as well. By doing so, the leak of the client's data implies an adversary has obtained the secret decryption key, thus the adversary can also obtain the cryptocurrency secret key by decrypting the published ciphertext. This allows automatic penalty, but unless using Fully Homomorphic

Encryption (FHE), the clouds cannot compute on the client's data (which is now encrypted). Yet, FHE is not practical to handle many computation that can be efficiently done using secret sharing based MPC. Similarly, generic all-or-nothing transform (ANOT) [38] and similar schemes cannot be applied because they do not have the algebraic properties that allow the clouds to compute on the client's data after transformation. How to bundle the client's data with the clouds' secret key while still allow computing on the client's data is not trivial.

The second challenge we experienced was how to differentiate an authorized reconstruction of the user's data and an unauthorized one. The difference is, if the client requests to reconstruct its data, the shares should only reveal the data but not the clouds' cryptocurrency secret key; if an external adversary somehow has gathered all shares and recovers the client data, the clouds' secret key must be revealed from the shares to enforce the penalty. Since what we do is to bundle two secrets (the client's data and the clouds' secret key) into one set of shares, readers who are familiar with multi-secret sharing [10] might think the above idea can be easily realized by using multi-secret sharing. However, what a multi-secret sharing scheme can do is one of the following two (but not both): either each secret can be recovered from a different subset of shares, or certain subsets of shares can recover all the secrets. In the former case, it implies that it is possible to leak the data without leaking the secret key (or leak the key when the client's data is not leaked). In the latter case, it is simply "all or nothing" in all situations. Clearly, both are not suitable for our purpose.

Other minor technical challenges include how to ensure integrity of the data stored in the clouds and the computation results, and how to bundle the client's data and clouds' secret key when the parties do not trust each other.

Our Contributions In this paper, we present a scheme called Authenticated Secret Sharing with Penalty (ASSP). The scheme is designed for the scenario in which a client outsources its data and MPC computation to multiple non-colluding clouds, and wants to enforce a penalty automatically if the data is compromised by an external adversary. Informally, the security properties of ASSP are the following: (1) the privacy and integrity of an honest client's data are preserved, as long as there is at least one honest cloud; (2) for an honest cloud, its deposit is safe even when the client and all other clouds are corrupted; (3) in the cases of all the clouds are corrupted by an external adversary, the client's data is leaked, and at the same time the cryptocurrency secret key must also be leaked (in order to enforce the penalty). We have also implemented ASSP and evaluated its practicality.

2 Technical Overview

The technical road map is depicted in Fig. 1. The two main cryptographic components are an authenticated secret sharing scheme and a set of protocols.

The authenticated secret sharing scheme is over the ring Z_N, where $N = p \cdot q$ is an integer with two large prime factors p and q. By the Chinese Remainder Theorem (CRT) [39], there is a ring isomorphism $Z_N \cong Z_p \times Z_q$. As a conse-

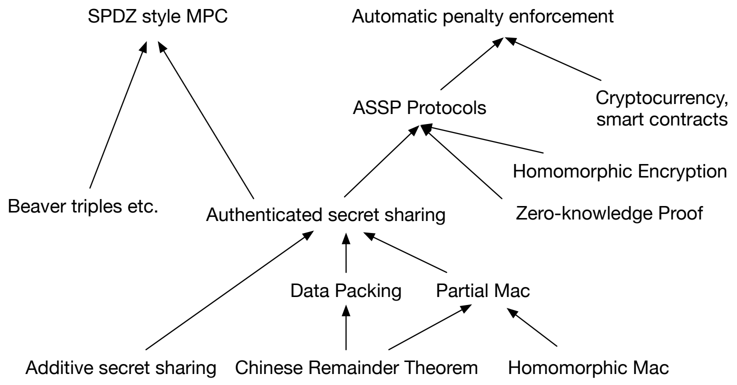

Fig. 1: Technical Road Map

quence, each element in Z_N can be viewed equivalently as a 2-vector (s_1, s_2) where $s_1 \in Z_p$ and $s_2 \in Z_q$. Thus we can pack the client's data and the clouds' secret key together, by using the Z_p slot in the vector to hold the client's data and the Z_q slot to hold the key. After mapping the vector $(s_1, s_2) \rightarrow S \in Z_N$, we can then split S into n additive shares easily. The secret shares can also be authenticated with homomorphic MACs, with a small technicality that only the client's data is authenticated (partial MAC). The resulted authenticated shares are compatible with SPDZ style MPC. Since the isomorphism is addition and multiplication preserving, any operation performed on the shares (in Z_N) will be passed to the client's data (now also shared but in Z_p). So, when the clouds performs MPC over the shares, they are actually doing the same computation over the client's data. This addresses the first major challenge we mentioned earlier.

Since we use Z_N for the shares, we can use the same parameters p, q, N for the Paillier encryption scheme [36]. The plaintext domain of the resulted Paillier instance is also Z_N, which allows us to conveniently design secure protocols for manipulating the shares. In ASSP, the clouds and the client run the Setup protocol in the setup phase, then the client can run the Share protocol to outsource its data to the clouds, at the same time bundle its data with the clouds' cryptocurrency secret key. We have two different protocols for reconstructing the secret(s) from the shares: one is the Reconstruct protocol that is an interactive protocol between the client and the clouds, and one is the Extract protocol that is non-interactive. The Reconstruct protocol recovers only the client's data and is the one used in the authorised reconstruction case; the Extract protocol recovers both the client's data and the clouds' key, and can be used by the external adversary in the unauthorised reconstruction case. This addresses the second major challenge.

3 Preliminary

3.1 Notations

For a set X, we denote by $x \overset{R}{\leftarrow} X$ the process of choosing an element x from the set X uniformly at random. In this paper, we will use different notations for plain secret shares and authenticated secret shares. A value a, when being shared in a secret sharing scheme, produces n plain shares $[\![a]\!]_1, \dots, [\![a]\!]_n$. The authenticated share $\langle a \rangle_i = ([\![a]\!]_i, [\![m(a)]\!]_i)$ where $m(a)$ is a MAC of a, and $[\![m(a)]\!]_i$ is a share of the MAC. We may, when it is clear from the context, omit the subscription i and just use $[\![a]\!]$ and $\langle a \rangle$. We will use $\star$ as a wildcard to denote an unnamed value when its value is not important. For example in $\mathsf{Enc}_{pk}(m, \star)$, we use $\star$ to denote some random number whose value we do not care.

3.2 Secret Sharing

Many MPC schemes (e.g. [13, 11, 17, 16]) use a simple (n, n)-secret sharing as a fundamental building block. The secret sharing scheme works as the following in an additive group $\mathbb{G}$:

- Share: Given a secret $s \in \mathbb{G}$, for $1 \leq i \leq n-1$, set $[\![s]\!]_i \overset{R}{\leftarrow} \mathbb{G}$, then set $[\![s]\!]_n = s - \sum_{i=1}^{n-1} [\![s]\!]_i$.
- Recon: Given all n shares, output the secret $s = \sum_{i=1}^{n} [\![s]\!]_i$.

3.3 Homomorphic MAC

Here we briefly describe the MAC scheme used in SPDZ [17] and its successors. In the n-party setting, the parties use the additive secret sharing scheme described in section 3.2 over a finite field $\mathbb{F}$. Each party chooses a random MAC key $\alpha_i \in \mathbb{F}$, and the global MAC key is $\alpha = \sum_{i=1}^{n} \alpha_i$. The MAC of a value $a \in \mathbb{F}$ is $m(a) = \alpha \cdot a$. The MAC is additively homomorphic, i.e. $m(a)+m(b) = \alpha \cdot (a+b) = m(a+b)$. The MAC is shared across the parties so that each party has the authenticated share of a that is $\langle a \rangle_i = ([\![a]\!]_i, [\![m(a)]\!]_i)$. The authenticity of the value can be checked by checking whether $\Delta = a \cdot \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n} [\![m(a)]\!]_i$ is 0. It is also possible to check multiple authenticated shares in one go by taking a random linear combination of Δ for each value and check whether the result is 0.

3.4 Paillier Cryptosystem

The Paillier encryption scheme is a public key scheme based on Decisional Composite Residuosity problem [36]. The scheme $\mathcal{PE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as follows:

- $\mathsf{Gen}(1^k)$: Take as input a security parameter k, generate two primes p, q (size determined by k). Compute $N = pq$. The public key $pk = N$ and the private key $sk = (N, p, q)$.

– $\mathsf{Enc}_{pk}(m, r)$: Take as input a message $m \in \mathbb{Z}_N$ and a public key, with a uniform $r \xleftarrow{R} Z_N^*$, the ciphertext $c := (1 + N)^m \cdot r^N \bmod N^2$.
– $\mathsf{Dec}_{sk}(c)$: Take as a ciphertext $c \in Z_{N^2}$ and a private key, the decrypted plaintext $m := \frac{(c^{\phi(N)} \bmod N^2)-1}{N} \cdot \phi(N)^{-1} \mod N$

The Paillier encryption scheme is additively homomorphic. We denote homomorphic addition by $\boxplus$ and homomorphic multiplication (with a constant) by $\boxdot$. At a high level, we can express the homomorphic operations as the following:

– Homomorphic Addition: Given two ciphertexts $c = \mathsf{Enc}_{pk}(m, r)$ and $c' = \mathsf{Enc}_{pk}(m', r')$, then $c_{add} = c \boxplus c' = ((1 + N)^m \cdot r^N) \cdot ((1 + N)^{m'} \cdot r'^N) = (1 + N)^{m+m'} \cdot (rr')^N = \mathsf{Enc}_{pk}(m + m', rr')$.
– Homomorphic Multiplication with a Constant: Given a ciphertext $c = \mathsf{Enc}_{pk}(m, r)$ and a constant a, then $c_{mult} = a \boxdot c = ((1 + N)^m \cdot r^N)^a = (1 + N)^{am} \cdot (r^a)^N = \mathsf{Enc}_{pk}(a \cdot m, r^a)$.

3.5 Zero-Knowledge Proof of Knowledge

A zero-knowledge proof of knowledge (ZKPoK) is a protocol between a prover and a verifier. Informally, the goal of the protocol is for the prover to convince the verifier that he knows a solution of a hard-to-solve problem. The protocol should be (1) *complete*: an honest prover who knows a solution can convince the verifier successfully; (2) *sound*: if the prover does not know any solution, then he will fail to convince the verifier (with an overwhelming probability); and (3) *zero-knowledge*: no information other than the fact the prover knows a solution is leaked. Formally, let L be an NP-language, and $R : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}$ be a polynomial time computable binary relation such that for all x and a witness w, $R(x, w) = 1$ *iff* $x \in L$. A ZKPoK protocol allows a prover to convince the verifier that it knows a witness w such that $R(x, w) = 1$ for some relation R and public x. It can be captured by an ideal functionality as in Figure 2.

Ideal Functionality $\mathcal{F}_{zk}^R$ [23]

The functionality interacts with two parties P and V. When receiving $(\mathbf{prove}, x, w)$ from P and $(\mathbf{verify}, x')$ from V, it does the following: If $x = x'$ then outputs $(\mathbf{proof}, R(x, w))$ to V, otherwise ignores the messages.

Fig. 2: The Ideal Functionality $\mathcal{F}_{zk}^R$

In ASSP, we use ZKPoK protocols to prove various relations among parameters, plaintexts and ciphertexts in the Paillier cryptosystem, so that a malicious party cannot cheat. In the following, we provide a high level description of the ZKPoK protocols. We use the notation $\{(w_1, \ldots, w_n) | st_1 \wedge \cdots \wedge st_m\}$ to describe the protocols. In the notation, all values $(w_1, \ldots, w_n)$ in the first part before $|$ is the secret knowledge possessed by the prover, and the part after $|$ is the statements the prover tries to prove. All values in the statement part excepts $(w_1, \ldots, w_n)$ are public.

– R_N: Proof of a Paillier public key N that was generated correctly.

$$\{(p,q) \mid N = pq \wedge p, q \text{ are primes}\}$$

– R_P: Proof of c is a proper Paillier ciphertext of a public value m.

$$\{(r) \mid c = \mathsf{Enc}_{pk}(m, r)\}$$

– R_M: Proof that a Paillier ciphertext c_3 encrypts the product of the plaintexts encrypted in Paillier ciphertexts c_1 and c_2.

$$\{(m_1, m_2) \mid \exists r_1, r_2, r_3 : c_1 = \mathsf{Enc}_{pk}(m_1, r_1) \wedge c_2 = \mathsf{Enc}_{pk}(m_2, r_2) \\ \wedge c_3 = \mathsf{Enc}_{pk}(m_1 \cdot m_2, r_3)\}$$

– R_R: Proof of a Paillier ciphertext c encrypts a plaintext whose value is in the range (L, H).

$$\{(x) \mid \exists r : \ c = \mathsf{Enc}_{pk}(x, r) \wedge L < x < H\}$$

– R_S: Proof of the plaintext encrypted in a Paillier ciphertext c_2 is in the form $s \cdot x_1 + r$, where x_1 is the plaintext encrypted in a Paillier ciphertext c_1 and s is a secret scalar committed in $P = s \cdot G$. Here G is a generator of an eclipse curve group $\mathbb{G}$:

$$\{(s, r) \mid \exists r_1, r_2 : P = s \cdot G \wedge c_1 = \mathsf{Enc}_{pk}(x_1, r_1) \\ \wedge c_2 = \mathsf{Enc}_{pk}(s \cdot x_1 + r, r_2)\}$$

Remark. All the above ZKPoK protocols exist in the literature. Specifically, R_N is from [12], [37]; R_p and R_M are from [15]; R_R is from [34]; and R_S is from [21].

3.6 Cryptocurrency Transactions

Here we briefly review the necessary background on cryptocurrency transactions. Loosely speaking, in a cryptocurrency system, funds are stored in addresses that are derived from public keys. Transferring funds from address A to address B is realized by publishing a valid transaction such that the transaction is verified and accepted by all parties maintaining the cryptocurrency through a global consensus protocol. In this section, we use Bitcoin [1] as an example, but the features we explain in this section are widely supported by other major cryptocurrencies, e.g. Etheruem [3], Zcash [4], albeit differences in implementation. A Bitcoin transaction has the following fields [2]:

– List of inputs: Specify where the funds come from. Each input is a reference to an output address from a previous transaction (Unspent Transaction Output, UTXO for short). Each input also has a ScriptSig field. This field contains the signature(s) of the current transaction, signed under the secret key(s) required to authorize the spending of the unspent balance in the UTXO address. Multiple inputs can be listed in one transaction, and the sum of unspent balance in them will be combined.

– List of outputs: Specify where the funds go and how to distribute them. Each output contains a ScriptPubKey filed that specifies the address the fund will go to and rules of authorizing spending from the output address, and a Value field that is the amount of fund going to the address. The address is usually specified in terms of one public key or a set of public keys (multi-sig). The subsequent transaction using this output address as input will need to provide the signatures matching the public key(s) specified in ScriptPubKey.
– lock_time: Specify the block number or timestamp at which this transaction becomes valid.

When specifying an output of a transaction, it is possible to use a multi-sig address by specifying in ScriptPubKey a set of n public keys and requiring m-out-of-n ($m \leq n$) valid signatures from distinct matching public keys to authorize the spending.

```
ScriptPubkey : <m> <A pubkey> [B pubkey] [C pubkey...]
               <n> OP_CHECKMULTISIG
```

Then to spend the fund in this multi-sig address, the subsequent transaction must specify this multi-sig address as input and include at least m signatures:

```
ScriptSig: OP_0 <A sig> [B sig] [C sig...]
```

In Bitcoin, one can specify in ScriptPubkey a temporal condition using the CheckLockTimeVerify op code, which was introduced in 2015 to Bitcoin by BIP65 [42]. For example:

```
ScriptPubkey : OP_IF <time t> OP_CHECKLOCKTIMEVERIFY DROP
                     <A pubkey> OP_CHECKSIGVERIFY
               OP_ELSE
                   <m> <A pubkey> [B pubkey] [C pubkey...]
                   <n> OP_CHECKMULTISIG
               OP_ENDIF
```

The script says that before time t, m out of n signatures are required to spend the funds from this output, and after time t, only A's signature is required to spend the funds. In the script, The CheckLockTimeVerify op code is provided with the time t. It checks the lock_time field of the transaction that attempts to spend funds from this output against t. Since a transaction with lock_time $= t_1$ can only be added into the blockchain after t_1, the comparison ensures current time $\geq t_1 \geq t$ for the IF condition to hold, and then one signature is required. If t is greater, then the condition fails, and any attempt to spend the funds will go to the ELSE branch and needs to provide multiple signatures.

3.7 Distributed Key Generation

In ASSP, we requires all clouds to jointly generate an ECDSA public/secret key pair such that each cloud holds a share of the secret key. This can be done using a Distributed Key Generation (DKG) protocol such as the one in [33]. Abstractly, a DKG protocol realize the ideal functionality in Figure 3.

Functionality $\mathcal{F}_{\text{KeyGen}}$

The ideal functionality works with parties $P_1, \cdots, P_n$, as follows:

– **KeyGen**: Upon receiving (KeyGen, $\mathbb{G}, G, p_e$) from all parties $P_1, \cdots, P_n$, where $\mathbb{G}$ is an Elliptic-curve group of order p_e with generator G:
  - Generate an ECDSA key pair (Q, s) by choosing a random $s \overset{R}{\leftarrow} \mathbb{Z}^*_{p_e}$ and computing $Q = s \cdot G$. Q is the public key and s is the secret key. Then, store $(\mathbb{G}, G, p_e, s)$.
  - Send $(Q, [\![s]\!]_i)$ to each $P_1, \cdots, P_n$ such that $[\![s]\!]_1, \cdots, [\![s]\!]_{n-1} \overset{R}{\leftarrow} \mathbb{Z}^*_{p_e}$ and $[\![s]\!]_n = s - \sum_{i=1}^{n-1} [\![s]\!]_i$.
  - ignore future calls to **KeyGen**.

Fig. 3: The Ideal Functionality $\mathcal{F}_{\text{KeyGen}}$

4 Security Model

4.1 Participants and Adversary Model

The application scenario of ASSP is that a client wants to outsource its data and computation to n non-colluding clouds. We consider an external adversary who can corrupt the client and the clouds. We consider the adversary to be *malicious* and *adaptive*. Here an adaptive adversary is more adequate in our scenario to capture data leakage. With a static adversary who chooses the parties to corrupt prior to the start, either the data is leaked at the very beginning (if the adversary corrupts all clouds), or there is no data leakage at all (if the adversary does not corrupt all clouds). However in reality, any number of parties can be compromised at any point in time during the life-cycle of the outsourced data.

4.2 Ideal Functionality

Informally, the security properties of ASSP are as the following: (1) the privacy and integrity of an honest client's data are preserved, as long as there is at least one honest cloud; (2) for an honest cloud, its secret key share is private even when the client and all other clouds are corrupted; (3) in the cases of all the clouds are corrupted by external attackers, the client's data is leaked, and at the same time the secret key shared among clouds must also be leaked (in order to enforce the penalty). Formally, the security of ASSP is defined using the ideal/real paradigm [22], and is captured by an ideal functionality as shown in Figure 4.

Note that the adversary can get the clouds' key shares in two ways: first, whenever a cloud S_i is compromised, its key share is leaked to the adversary; second, when the last honest cloud is compromised, the adversary is given the secret key. The first captures the worst case in which a cloud is fully compromised and/or colludes with the adversary. In this case, the adversary knows the cloud's key share. The ideal functionality captures this by leaking the key share to the adversary when the cloud is compromised, and its definition also ensures that even with the knowledge of the leaked key share, the adversary learns nothing about the client's data or honest clouds' key shares. The second one captures the case where a weaker adversary who is not able to gain access to

Functionality $\mathcal{F}_{\text{ASSP}}$

The functionality maintains a dictionary Val (for the client's data) and an n-vector K (for the clouds' secret key share), a state variable $ST \in \{-1, 0, 1\}$ whose initial value is -1. The parties are n clouds $S_1, \ldots, S_n$ and a client C. An updatable set H denotes the set of honest parties. All parties not in H are corrupted.

– Setup:
  - On receiving (Setup, p) from the client, check ST and p. If $ST = -1$ and p is prime, set $ST = 0$, and initialize Val with Z_p as the domain of all values to be stored in it. Ignore the message otherwise.
  - On receiving $(\mathit{Setup}, [\![s]\!]_i, Q_i)$ message from cloud S_i, check ST, if $ST = 0$, then check whether $K[i]$ is empty, if so, then check whether $[\![s]\!]_i$ is the secret key corresponding to Q_i, if so set $K[i] = [\![s]\!]_i$. Ignore the message otherwise.
  - After receiving $(\mathit{Setup}, \mathit{go})$ from all parties, set $ST = 1$.

– Share
  - On receiving $(\mathit{Share}, \mathit{id}, x)$ from the client and $(\mathit{Share}, \mathit{id})$ from all clouds, check whether $ST = 1$ and $\mathit{id} \notin \mathit{key}(\mathsf{Val})$, if so put (id, x) in the dictionary. Ignore the message otherwise. If all clouds are corrupted, the functionality additionally sends (id, x) to the adversary.

– Compute: check whether $ST = 1$, if not then ignore the message.
  - On receiving command $(\mathit{add}, \mathit{id}_1, \mathit{id}_2, \mathit{id}_3)$ from all clouds, the functionality retrieves $v_1 \leftarrow \mathsf{Val}[\mathit{id}_1]$ and $v_2 \leftarrow \mathsf{Val}[\mathit{id}_2]$ and stores $\mathsf{Val}[\mathit{id}_3] = v_1 + v_2 \bmod p$.
  - On receiving command $(\mathit{add}, \mathit{id}_1, e, \mathit{id}_3)$ from all clouds, where e is a constant, the functionality retrieves $v_1 \leftarrow \mathsf{Val}[\mathit{id}_1]$ and stores $\mathsf{Val}[\mathit{id}_3] = v_1 + e \bmod p$.
  - On receiving command $(\mathit{multiply}, \mathit{id}_1, \mathit{id}_2, \mathit{id}_3)$ from all clouds, the functionality retrieves $v_1 \leftarrow \mathsf{Val}[\mathit{id}_1]$ and $v_2 \leftarrow \mathsf{Val}[\mathit{id}_2]$ and stores $\mathsf{Val}[\mathit{id}_3] = v_1 \cdot v_2 \bmod p$.
  - On receiving command $(\mathit{multiply}, \mathit{id}_1, e, \mathit{id}_3)$ from all clouds, where e is a constant, the functionality retrieves $v_1 \leftarrow \mathsf{Val}[\mathit{id}_1]$ and stores $\mathsf{Val}[\mathit{id}_3] = v_1 \cdot e \bmod p$.

– Reconstruct:
  - On receiving $(\mathit{Reconstruct}, \mathit{id})$ from all parties, check whether $ST = 1$, if not then ignore the message. Check whether $\mathit{id} \in \mathit{key}(\mathsf{Val})$, if not, ignore the message. Otherwise, retrieve $x = \mathsf{Val}[\mathit{id}]$. If at least one cloud is not corrupted, send x to client, otherwise wait x' from the adversary, sends x' to client.

– Corrupt:
  - On receiving $(\mathit{corrupt}, i)$ from the adversary ($i = 0$ for the client, $i = 1, \cdots, n$ for each cloud S_i), if the party is in H, remove it from H. Stop sending and receiving messages from this party. Instead all messages to and from this party will be sent to and received from the adversary. When corrupting a cloud S_i, take $[\![s]\!]_i$ from K and send it to the adversary. If S_i is the last honest cloud, send Val and $s = \sum_{i=1}^{n} K[i]$ to the adversary.

– Abort:
  - On receiving (Abort) from any party, send abort to all parties and terminate.

Fig. 4: The Ideal Functionality $\mathcal{F}_{\text{ASSP}}$

the secret key share when compromising a cloud. For example, the key share can be stored in a cold storage offline. We want to make sure that even if the adversary is weaker and cannot obtain the key share at the time of compromising each cloud, the penalty is still enforceable if all clouds have been compromised. The ideal functionality captures this and ensures when the last honest cloud is compromised and there is no way to keep the data in the shares secure, the shared key must be still leaked with the client's data to the adversary, no matter whether the adversary has obtained the key shares in the past.

5 The Construction

5.1 Data Packing

As mentioned earlier, we will use the Chinese Remainder Theorem (CRT) to pack the client's data and the clouds' secret key together. We will work with a special case of CRT over the ring Z_N, in which $N = p \cdot q$ where p and q are two large primes. For convenience, in all the following text we will use the Z_p slot for the client's data and Z_q slot for the clouds' secret key. We have n clouds and each of them has a share of the secret key generated from DKG protocol. Therefore, we require the size of the prime factors to be larger than that of $\sum_{i=1}^{n} [\![s]\!]_i$ so that the Z_q slot is large enough for the secret key. The size also needs be large enough to accommodate the data item and to make it infeasible to factor N (with regard to a security parameter k).

By CRT, $Z_N \cong Z_p \times Z_q$ and we have two mappings to pack and unpack data: $\mathsf{crt} : Z_p \times Z_q \to Z_N$ and $\mathsf{crt}^{-1} : Z_N \to Z_p \times Z_q$. The mapping for packing data is defined as the following $\mathsf{crt}(x_1, x_2) = \lambda \cdot q \cdot x_1 + \mu \cdot p \cdot x_2 \bmod N$ for any $x_1 \in Z_p$ and $x_2 \in Z_q$, where λ and μ are constants that can be found by the Extended Euclidean algorithm such that $\lambda \cdot q + \mu \cdot p = 1$. The unpacking mapping is defined as $\mathsf{crt}^{-1}(x) = (x \bmod p, x \bmod q)$. Both mappings can be computed efficiently when the factorization of N is known.

In this paper, when using $x \in Z_N$ we will often write it in the equivalent CRT form $\mathsf{crt}(x_1, x_2)$ to make it is easier to see the computation performed on the vector components. As we explained, the isomorphism is addition and multiplication preserving. Thus for $x, y \in Z_N$ such that $x = \mathsf{crt}(x_1, x_2), y = \mathsf{crt}(y_1, y_2)$ we have $x + y = \mathsf{crt}(x_1, x_2) + \mathsf{crt}(y_1, y_2) = \mathsf{crt}(x_1 + y_1, x_2 + y_2)$ and $x \cdot y = \mathsf{crt}(x_1, x_2) \cdot \mathsf{crt}(y_1, y_2) = \mathsf{crt}(x_1 \cdot y_1, x_2 \cdot y_2)$. In ASSP, we will often use the properties to manipulate the data in the slots.

5.2 Partial MAC Scheme

As discussed earlier, the client needs to protect the integrity of its data and computation result. To achieve the goal, we use a homomorphic MAC scheme modified from the one mentioned in Section 3.3. To authenticate its data, the client holds the MAC key, generates and shares the MAC when sharing the data, and verifies the MAC when reconstructing the data from shares. One complication is that the client's data is packed with the clouds' secret key. However, when generating and verifying the MAC, the client should know only its own data in the Z_p slot, but nothing about the clouds' secret keys in the Z_q slot. Therefore we cannot use the MAC scheme in Section 3.3 directly, but have to use a partial MAC that concerns only the data in the Z_p slot. The partial MAC scheme is as the following:

– Key generation: Given $N = p \cdot q$ that is an integer as described in Section 5.1, sample a MAC key $\alpha \overset{R}{\leftarrow} Z_N^*$. Equivalently, $\alpha = \mathsf{crt}(\alpha_1, \alpha_2)$ in the CRT form.

- MAC generation: For $e = \mathsf{crt}(d, \star) \in Z_N$, its MAC is created on $e' = \mathsf{crt}(d, 0)$ as $m(e) = \alpha \cdot e' = \mathsf{crt}(d, 0) \cdot \mathsf{crt}(\alpha_1, \alpha_2) = \mathsf{crt}(\alpha_1 \cdot d, 0)$.
- MAC verification: For $e \in Z_N$ and its MAC $m(e) \in Z_N$, verify $\alpha \cdot (e \cdot \mathsf{crt}(1, 0)) \overset{?}{=} m(e)$.

Essentially, when creating the MAC, the client only authenticates the data d in the Z_p slot, and when verifying the MAC, the client only checks whether the value in the Z_p slot is intact. Since the client only cares about d, this is sufficient. As a remark, this partial MAC scheme is not intended for general message authentication.

The additive homomorphic property of the partial MAC scheme is easy to see. We can see that for any $x = \mathsf{crt}(d_1, \star)$, $y = \mathsf{crt}(d_2, \star)$ and their MACs $m(x) = \mathsf{crt}(\alpha_1 \cdot d_1, 0)$, $m(y) = \mathsf{crt}(\alpha_1 \cdot d_2, 0)$, it holds that $m(x) + m(y) = \mathsf{crt}(\alpha_1 \cdot (d_1 + d_2), 0)$, which is the valid partial MAC for $x + y = \mathsf{crt}(d_1 + d_2, \star)$.

5.3 Authenticated Shares

In ASSP, the client shares $e = \mathsf{crt}(d, \star) \in Z_N$ using the simple (n, n) additive secret sharing scheme described in Section 3.2. That is, the shares of e are n random looking numbers in Z_N, denoted by $[\![e]\!]_1, \ldots, [\![e]\!]_n$, which sum to e. The client also shares $m(e)$, the partial MAC of e, using the same (n, n) additive secret sharing scheme. Then the authenticated share is a pair $\langle e \rangle = ([\![e]\!], [\![m(e)]\!])$. Each cloud will obtain one share only. Later when reconstructing the data, the client can check whether the data in the shares is intact:

- Check: take $\langle e \rangle_1, \cdots, \langle e \rangle_n$ as input, recover $e \cdot \mathsf{crt}(1, 0) = [\![e]\!]_1 \cdot \mathsf{crt}(1, 0) + \cdots + [\![e]\!]_n \cdot \mathsf{crt}(1, 0)$ and $m(e) = [\![m(e)]\!]_1 + \cdots + [\![m(e)]\!]_n$. Output 1 if $m(e)$ is a valid MAC of e, 0 otherwise.

The authenticated shares guarantee perfect privacy and computational integrity. Informally, perfect privacy means that any subset of less than n shares reveals no information about the value being shared and the MAC key being used. This holds even when the adversary has unbounded computational power. Computational integrity means that it is infeasible for a Probabilistic Polynomial Time (PPT) adversary who has obtained any subset of less than n shares to modify the shares it obtained, and later pass the MAC verification. We have the following theorem regarding the security of the authenticated shares:

Theorem 1. *Let $S = \{\langle e \rangle_1, \cdots, \langle e \rangle_n\}$ be the set of n authenticated shares of e, $S' \subset S$ be an arbitrary proper subset of S. The following hold:*

1. *Perfect Privacy: For any adversary $\mathcal{A}$, given S', $\mathcal{A}$ learns nothing about e and the MAC key α. That is, we have $Pr[\mathcal{A}(S') \rightarrow e] = Pr[\mathcal{A}(\perp) \rightarrow e]$ and $Pr[\mathcal{A}(S') \rightarrow \alpha] = Pr[\mathcal{A}(\perp) \rightarrow \alpha]$;*
2. *Computational Integrity: Assuming the factorization problem is hard, for any PPT adversary $\mathcal{A}$, given S', $\mathcal{A}$ cannot modify the shares and pass the MAC check, i.e. $\mathcal{A}(S') \rightarrow S''$ such that $S' \neq S''$ and let $\tilde{S} = S'' \cup (S - S')$,*

$Pr[\mathrm{Check}(\tilde{S}) = 1] \leq negl(k)$, where $negl(\cdot)$ is a negligible function and k is a security parameter.

Proof. The first part of the theorem follows directly from the perfect privacy property of the (n, n) secret sharing scheme.

For the second part, without loss of generality, assume $\langle e \rangle_1 \in S'$ and S'' is obtained from S' by replacing $\langle e \rangle_1$ with $\langle e \rangle_1' = (\llbracket e \rrbracket_1 + \delta_1, \llbracket m(e) \rrbracket_1 + \delta_2)$. This is general enough because when checking the shares, the sums of the shares (of the data and the MAC) are used, therefore we do not care which shares are modified and can treat the variation as coming from one share. It is clear that given $\tilde{S}$, one can recover $e' = \llbracket e \rrbracket_1' + \llbracket e \rrbracket_2 + \cdots + \llbracket e \rrbracket_n = e + \delta_1$ and $m(e)' = \llbracket m(e) \rrbracket_1' + \llbracket me \rrbracket_2 + \cdots + \llbracket m(e) \rrbracket_n = m(e) + \delta_2$.

We claim that if the modified shares pass the MAC verification with a non-negligible probability, then we can find the factors of N with a non-negligible probability. Passing the verification means:

$$\alpha \cdot (e + \delta_1) \cdot \mathsf{crt}(1, 0) = m(e) + \delta_2. \tag{1}$$

Recall that in their CRT forms $\alpha = \mathsf{crt}(\alpha_1, \alpha_2), e = \mathsf{crt}(d, \star), m(e) = \mathsf{crt}(\alpha_1 \cdot d, 0)$ and we can rewrite $\delta_1 = \mathsf{crt}(\delta_1', \delta_1''), \delta_2 = \mathsf{crt}(\delta_2', \delta_2'')$, then Equation 1 can be rewritten as:

$$\mathsf{crt}(\alpha_1 \cdot (d + \delta_1'), 0) = \mathsf{crt}(\alpha_1 \cdot d + \delta_2', \delta_2''). \tag{2}$$

Since the CRT mapping is a bijection, equation (2) implies $\delta_2'' = 0$. Then there are two cases:

Case (1): $\delta_2 = 0 = \mathsf{crt}(0, 0)$. In this case, for equation (2) to hold, we must have $\alpha_1 \cdot (d + \delta_1') = \alpha_1 \cdot d$, which only holds when $\delta_1' = 0$. There is one trivial case $\delta_1 = \mathsf{crt}(0, 0)$, but in this case $S'' = S'$ which means the $\mathcal{A}$ has not modified any shares. We then show that $\mathcal{A}$ cannot find $\delta_1 = \mathsf{crt}(0, \delta_1'')$ with $\delta_1'' \neq 0$. If $\mathcal{A}$ can do this, then we can use $\mathcal{A}$ as a subroutine to find the factorization of N. To do so, we simply generate S' that contains up to $n - 1$ fake authenticated shares $\langle x \rangle_i = (\llbracket x \rrbracket_i, \llbracket m(x) \rrbracket_i)$ such that $\llbracket x \rrbracket_i$ and $\llbracket m(x) \rrbracket_i$ are random element from Z_N. S' is a valid input to $\mathcal{A}$, and given S' it will output S''. We can then extract $\delta_1 = \sum_{i \in S''} \llbracket x \rrbracket_i - \sum_{j \in S'} \llbracket x \rrbracket_j = \mathsf{crt}(0, \delta_1'') = \mu \cdot p \cdot \delta_1''$. We can then find $p = GCD(\delta_1, N)$ and q after obtaining p. The probability of success in finding the factors is η where η is the probability of $\mathcal{A}$ modifying the shares successfully.

Case (2): $\delta_2 = \mathsf{crt}(\delta_2', 0)$ for some $\delta_2' \neq 0$. Again, we can reduce this case to finding the factors of N. The process is similar to what we did in Case 1, except we now extract δ_2 and obtain q then p.

This completes our proof.

A careful reader may concern about the confidentiality of the data in the Z_q slot, which we said was the secret key, being leaked during the MAC verification process. Indeed, if we allow the client to see e directly then the client can learn the clouds' secret keys. However, later we will see that in the case of an authorised reconstruction, when receiving a request from the client, the clouds run a protocol with the client that will output $e \cdot \mathsf{crt}(1, 0) = \mathsf{crt}(d, 0)$ to the client. This effectively eliminates the keys in the Z_q slot, thus the keys will not be leaked in the verification process.

5.4 Penalty-or-refund Contract

In ASSP, each cloud pays a deposit. The deposits from the clouds will be held in escrow under a penalty-or-refund contract. The idea is, if the client's data is compromised before an agreed time point t, then the clouds will be penalized by losing their deposits and the client will get compensated from the deposits; if nothing happens, then after t, each cloud's deposit will get refunded. Using Bitcoin as an example, the above idea can be implemented by two transactions: one is signed by all the clouds and includes the penalty-or-refund contract, and the other is partially signed by the client. The contract and the transactions are depicted in Fig. 5. Below, we will explain what the parties should do and how the two transactions achieve what we want.

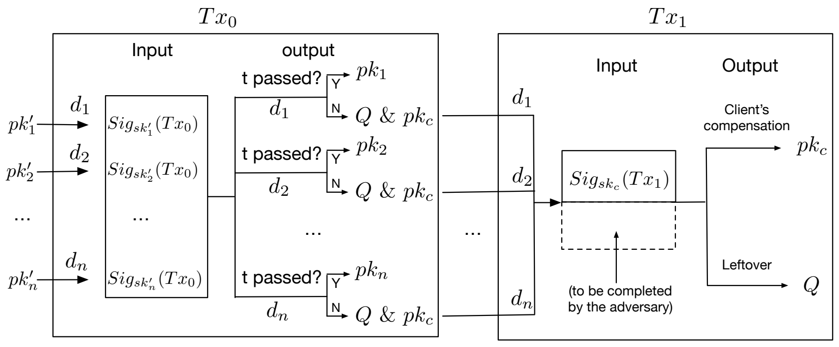

Fig. 5: The Penalty-or-refund Contract and the Transactions

1. The client and clouds agree on t as the time point when the outsourcing service ends, and d_i as the deposit for cloud S_i.
2. The clouds call $\mathcal{F}_{\text{KeyGen}}$ to jointly generate a public/secret key pair (Q, s), and each obtains $(Q, [\![s]\!]_i)$. Here Q is the public key, which is publicized.
3. Each cloud S_i must have an cryptocurrency balance greater than d_i, which is stored in address pk'_i, and holds the corresponding secret key sk'_i. Each S_i also generates a public/secret key pair (pk_i, sk_i) for the cryptocurrency, so that they can receive refund using pk_i as the address. The client generates a public/secret key pair (pk_c, sk_c) to receive the compensation.
4. All clouds jointly generates a transaction Tx_0, which has n inputs and n outputs (see Fig. 5). The deposit d_i will be taken from the input address pk'_i. Each output has a value d_i, and a script as follows:

```
ScriptPubkey : OP_IF <time t> OP_CHECKLOCKTIMEVERIFY DROP
                    pki OP_CHECKSIGVERIFY
               OP_ELSE
                    2 Q pkc 2 OP_CHECKMULTISIG
               OP_ENDIF
```

   The script says that after t, the amount d_i can be claimed by an transaction with a signature generated using the secret key corresponding to pk_i; but before t, claiming d_i requires two signatures: one by the secret key jointly generated by the clouds and one by the secret key sk_c owned by the client.

5. All clouds sign Tx_0 with its secret key sk'_j, then send Tx_0 to the blockchain.
6. The client waits until Tx_0 is confirmed. Then, the client produces a transaction Tx_1. Tx_1 has n inputs, corresponding to the n multisig branches in Tx_0, and takes all deposits over. Tx_1 has two outputs: one outputs the agreed amount of compensation to pk_c, and the leftover goes to Q (the public key jointly generated by the clouds).
7. The client then signs Tx_1 with sk_c, and publishes the partially signed transaction.

Note that Tx_1 is only partially signed, so it is not valid until someone co-signs it with s, the secret key jointly generated by the clouds. The key s is not known by anyone when it is generated, but will be leaked if the client's outsourced data is leaked. Therefore the adversary who obtained s can co-sign Tx_1 and send the fully signed transaction to the blockchain. Then the deposits from the clouds will be distributed according to the transaction. The adversary can further transfer the amount going to the address corresponding to the public key Q to any other address it controls, since it knows the secret key s. The adversary cannot generate a new transaction and change the distribution because the client's secret key is needed to authorize the new transaction. If no data leakage, then Tx_1 remains invalid. Then after t, the clouds can claim their deposits back.

5.5 Main Protocols in ASSP

We are now ready to present the main protocols in ASSP. In all the protocols, if a cheating behaviour is detected, the default behaviour of the honest parties is to abort and terminate the protocol execution, unless otherwise stated.

Setup protocol. The Setup protocol (Fig. 6) only needs to be run once by the client and the clouds. They jointly generate keys and parameters to be used later in other protocols.

As shown in Fig. 6, the protocol can be split roughly into two steps: client key generation and cloud side parameters generation. The client key generation step is straightforward, the client generates a Paillier key pair, a MAC key for share authentication, and CRT parameters (λ, μ). The client needs to prove to the clouds that the Paillier key pair is generated correctly.

Then the client and the clouds need to generate other parameters $(c_1, c_2, [\![\alpha]\!]_i, [\![\mathsf{crt}(0,s)]\!]_i)$. The parameters include c_1 and c_2 that are the Paillier ciphertexts of two CRT vectors of special forms $\mathsf{crt}(0,1)$ and $\mathsf{crt}(1,0)$. The first ciphertext c_1 will be used by the clouds later in the setup protocol to embed their share $[\![s]\!]_i$ of secret key s into the Z_q slot, in the form of $\mathsf{crt}(0,[\![s]\!]_i)$. The second ciphertext c_2 will be used by the clouds in the Reconstruct protocol to eliminate their secret keys from the shares. The client needs to provide $\mathsf{crt}(0,1)$ and $\mathsf{crt}(1,0)$ in ciphertexts because if given in plaintexts, they will allow the clouds to learn the factorization of N, which will make the whole scheme insecure. However, encryption gives the client an opportunity to cheat. To prevent a malicious client from cheating, the client needs to prove that the two ciphertexts are well formed and encrypt the correct values. Note that the proofs in Step 2.(2) prove that

$\underline{\mathsf{Setup}\text{ Protocol}}$

1 Client side key generation

(1) Given the security parameter and the number of clouds, the client generates a pair of large primes (p, q). Then the client computes $N = p \cdot q$ and sets $(pk, sk) = (N, (N, p, q))$ to be a public/secret key pair in the Paillier encryption scheme.

(2) The client proves that N is of the correct form to the clouds using the ZKPoK protocol for R_N.

(3) The client runs the extend Euclidean algorithm to find (λ, μ) such that $\lambda \cdot q + \mu \cdot p = 1$. The client keeps (λ, μ) private.

(4) The client generates a key for the partial MAC scheme $\alpha \overset{R}{\leftarrow} Z_N^*$, and stores it privately.

2 Cloud side parameters $(c_1, c_2, [\![\alpha]\!]_i, [\![\mathsf{crt}(0, s)]\!]_i)$ generation

(1) The client computes the following two ciphertexts $c_1 = \mathsf{Enc}_{pk}(\mathsf{crt}(0, 1), \star)$, $c_2 = \mathsf{Enc}_{pk}(\mathsf{crt}(1, 0), \star)$, and broadcasts (c_1, c_2) to the clouds. The clouds store them locally.

(2) The client proves that the plaintexts in c_1, c_2 are greater than 1 and less than N using ZkPoK for the R_R relation. The client then proves that the sum of the plaintexts in c_1, c_2 is $\mathsf{crt}(1, 1) = 1 \bmod N$ by showing that the ciphertext $c_3 = c_1 \boxplus c_2$ encrypts the value 1 using ZkPoK for the R_p relation. The client also proves that the product of the plaintexts in c_1, c_2 is $\mathsf{crt}(0, 0) = 0 \bmod N$. This is done by first broadcasting a ciphertext $c_4 = \mathsf{Enc}_{pk}(0, \star)$ and proving that it is a ciphertext of the value 0 using ZkPoK for the R_p relation, then using ZKPoK for R_M to prove the product relation among the plaintexts.

(3) For each cloud S_i, $i \in [1, n]$ who has a secret key share $[\![s]\!]_i$ such that $Q_i = [\![s]\!]_i \cdot G$ and $Q = Q_1 + ... + Q_n$:

(a) S_i chooses $r_i \overset{R}{\leftarrow} Z_N$, computes homomorphically $\hat{c}_i = ([\![s]\!]_i \boxdot c_1) \boxplus \mathsf{Enc}_{pk}(r_i, \star) = \mathsf{Enc}_{pk}(\mathsf{crt}(0, 1) \cdot [\![s]\!]_i + r_i, \star)$, sends $\hat{c}_i$ to the client.

(b) S_i proves to the client that the homomorphic operations were done correctly by showing $\hat{c}_i$ encrypts a plaintext of the form $\mathsf{crt}(0, 1) \cdot [\![s]\!]_i + r_i$ where $[\![s]\!]_i$ is the correct share for the secret key s generated jointly by clouds (the client has obtained the public key $Q = s \cdot G$ and verified the balance in the account and correctness of each Q_i, i.e., $Q = Q_1 + ... + Q_n$), and r_i is a random value known by S_i. Note that the cloud does not need to know the plaintext value of $\mathsf{crt}(0, 1)$, in the proof only its ciphertext c_1 is needed. This is done by using the ZKPoK protocol R_S relation.

(4) The client decrypts each $\hat{c}_i$ and obtains the plaintext $x_i = \mathsf{crt}(0, 1) \cdot [\![s]\!]_i + r_i = \mathsf{crt}(0, [\![s]\!]_i) + r_i$. Note that $[\![s]\!]_i < q$ thus $[\![s]\!]_i = \mathsf{crt}(\star, s_i)$ and thus $\mathsf{crt}(0, 1) \cdot [\![s]\!]_i = \mathsf{crt}(0, [\![s]\!]_i)$. The client then computes $[\![\mathsf{crt}(0, s)]\!]_0 = \sum_{i=1}^n x_i$ where $s = \sum_{i=1}^n [\![s]\!]_i$.

(5) The client stores $[\![\mathsf{crt}(0, s)]\!]_0$ computed in the last step privately. Each cloud S_i stores $[\![\mathsf{crt}(0, s)]\!]_i = -r_i$ privately.

(6) Client then shares α among clouds. Each cloud S_i chooses $r'_i \overset{R}{\leftarrow} Z_N$ and sends it to the client. The client computes $r_\alpha = \alpha - \sum_{i=1}^n r'_i$ and broadcasts it to the clouds. The first cloud S_1 sets its share $[\![\alpha]\!]_1 = r_\alpha + r'_1$ and for $2 \le i \le n$, the cloud S_i sets its share $[\![\alpha]\!]_i = r'_i$.

Fig. 6: The Setup Protocol in ASSP

the plaintexts in c_1 and c_2 are not $\mathsf{crt}(0, 0) = 0 \bmod N$ or $\mathsf{crt}(1, 1) = 1 \bmod N$, then the sum and the product of the two plaintexts are $\mathsf{crt}(1, 1)$ and $\mathsf{crt}(0, 0)$ respectively, therefore c_1 encrypts either $\mathsf{crt}(0, 1)$ or $\mathsf{crt}(1, 0)$ and c_2 encrypts the other. This is sufficient because the two slots in the the CRT vector are symmetric, and it does not matter which slot is used for the client's data or clouds' key. We designate the first slot for the data and the second slot for the keys only for sake of simplicity of presentation. From a cloud's point of view, it does not matter whether c_1 encrypts $\mathsf{crt}(0,1)$ or $\mathsf{crt}(1,0)$, as long as c_2 encrypts the other, which ensures the key it inputs using c_1 can be eliminated later in the Reconstruct protocol using c_2.

The parameter $[\![\alpha]\!]_i$ is a share of the client's MAC key α. It will be used by the clouds when performing certain MPC operations on the shares. The last parameter is a share of $\mathsf{crt}(0,s) = \mathsf{crt}(0, \sum_{i=1}^{n} [\![s]\!]_i)$ that contains the shared secret key generated from DKG functionality. The shares are computed in step 2.(3) – 2.(5). Note that, the value $\mathsf{crt}(0,s)$ is shared using an $(n+1, n+1)$-additive secret sharing system, and each party (the client and n clouds) holds a share. This is the only place we use $(n+1, n+1)$ sharing. Later the Share protocol generates (n, n) shares and only the clouds hold the shares.

Share Protocol

1 The client's input is $d \in Z_p$. The client first encodes d into $\mathsf{crt}(d,0)$ then computes $d' = \mathsf{crt}(d,0) + [\![\mathsf{crt}(0,s)]\!]_0$, and the partial MAC $m(e) = \alpha \cdot \mathsf{crt}(d,0)$, where $e = \mathsf{crt}(d,s)$ is the overall value to be shared.

2 Each cloud chooses two random numbers $r_i, r'_i \xleftarrow{R} Z_N$, and sends them to the client. The client computes $r_e = d' - \sum_{i=1}^{n} r_i$ and $r_m = m(e) - \sum_{i=1}^{n} r'_i$, broadcasts r_e, r_m to the clouds.

3 The first cloud S_1 obtains $[\![e]\!]_1 = r_e + r_1 + [\![\mathsf{crt}(0,s)]\!]_1$ and $[\![m(e)]\!]_1 = r_m + r'_1$. The other clouds S_i each obtains $[\![e]\!]_i = r_i + [\![\mathsf{crt}(0,s)]\!]_i$ and $[\![m(e)]\!]_i = r'_i$. For all clouds, they store the authenticated share $\langle e \rangle_i = ([\![e]\!]_i, [\![m(e)]\!]_i)$ where $e = \mathsf{crt}(d,s)$.

Reconstruct Protocol

1 Each cloud S_i inputs an authenticated share $\langle e \rangle_i = ([\![e]\!]_i, [\![m(e)]\!]_i)$, and computes $\tilde{c}_i = [\![e]\!]_i \boxdot c_2$. Then S_i sends $(\tilde{c}_i, [\![m(e)]\!]_i))$ to the client.

2 The client decrypts each $x_i = \mathsf{Dec}(\tilde{c}_i)$ and computes $e' = \sum_{i=1}^{n} x_i$. The client also computes $m(e) = \sum_{i=1}^{n} [\![m(e)]\!]_i$. The client checks if $m(e) \equiv \alpha \cdot e'$. If so, obtain $d = e' \bmod p$

Extract Protocol

1 An adversary who has compromised all clouds can obtain $[\![\mathsf{crt}(d_1,s)]\!]_1, \ldots, [\![\mathsf{crt}(d_1,s)]\!]_n$ and $[\![\mathsf{crt}(d_2,s)]\!]_1, \ldots, [\![\mathsf{crt}(d_2,s)]\!]_n$. The adversary in turn can obtain $\mathsf{crt}(d_1,s)$ and $\mathsf{crt}(d_2,s)$ by summing the shares up.

2 The adversary computes $t = \mathsf{crt}(d_1,s) - \mathsf{crt}(d_2,s) = \mathsf{crt}(d_1 - d_2, 0) = \lambda \cdot q \cdot (d_1 - d_2) + \mu \cdot p \cdot 0 = \lambda \cdot q \cdot (d_1 - d_2)$. Then the adversary can compute $q = GCD(t, N)$, $p = N/q$. As such, the client's data can be recovered by $d_1 = \mathsf{crt}(d_1,s) \bmod p$ (similarly for other values recovered from shares), and the secret keys can be recovered by $s = (\mathsf{crt}(d_1,s) \bmod q) \bmod p_e$ where p_e is the group order of the eclipse curve group.

Fig. 7: The Share, Reconstruct and Extract Protocols in ASSP

Share protocol. The Share protocol (Fig. 7) does two things: (1) it packs together the client's data and the shared secret key s into a CRT vector; (2) it creates authenticated shares of the vector.

To pack the data and the keys together, the client first embeds the data into the Z_p slot as $\mathsf{crt}(d,0)$. Then recall that in the Setup protocol, the client and the clouds already generated the shares of $\mathsf{crt}(0,s)$ where $s = \sum_{i=1}^{n} [\![s]\!]_i$ is shared among all clouds where each cloud holds a share $[\![s]\!]_i$. Using the homomorphic property of the shares, the client and clouds add them together and obtain the shares of $\mathsf{crt}(d,s)$. As we can see in Step 1 and 3, the client and all clouds input their shares $[\![\mathsf{crt}(0, s)]\!]_i$ $(0 \leq i \leq n)$. We can see that the sum of the newly produced shares $\sum_{i=1}^{n}[\![e]\!]_i = d' - \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} r_i + \sum_{i=1}^{n}[\![\mathsf{crt}(0, s)]\!]_i = \mathsf{crt}(d, 0) + \mathsf{crt}(0, s) = \mathsf{crt}(d, s)$, so each $[\![e]\!]_i$ is indeed a share of $\mathsf{crt}(d, s)$. In the sharing process, each cloud contributes a random number so the values exchanged in the protocol do not leak information about the data and the keys. The partial MAC is shared in a similar way.

Reconstruct Protocol. The Reconstruct protocol (Fig. 7) ensures that when the client wants to reconstruct a piece of data, only its data d is revealed. This is achieved in the protocol by a homomorphic operation performed on each cloud, using $c_2 = \mathsf{Enc}_{pk}(\mathsf{crt}(1, 0), \star)$ which was obtained by the cloud in the Setup protocol. The client can only obtain $\mathsf{crt}(d, 0)$.

Extract Protocol. The Extract protocol (Fig. 7) allows an external adversary who has compromised all the clouds, to recover the client's data and the clouds' secret key. The Extract protocol is non-interactive thus the adversary can run it locally after compromising all the clouds.

5.6 MPC Based on ASSP

One primary design goal of ASSP is to be MPC friendly, so that after the client shared its data to the clouds, the clouds can compute on the shares when instructed by the client. In this section we discuss how to perform MPC using the shares in ASSP.

Recall that for each piece of the client's data, the clouds hold authenticated shares $\langle e\rangle = ([\![e]\!], [\![m(e)]\!])$. Both parts in the authenticated share are homomorphic. This enables us to use the same protocols as in the online phase of SPDZ [17], for MPC over shares. The SPDZ online protocols can perform the following computation locally on the shares:

- $\langle x + y\rangle \leftarrow \langle x\rangle + \langle y\rangle$: Each cloud S_i takes $\langle x\rangle_i = ([\![x]\!]_i, [\![(m(x)]\!]_i)$, $\langle y\rangle_i = ([\![y]\!]_i, [\![m(y)]\!]_i)$ as the input, locally computes $[\![x + y]\!]_i = [\![x]\!]_i + [\![y]\!]_i$, $[\![m(x + y)]\!]_i = [\![m(x)]\!]_i + [\![m(y)]\!]_i$, and sets $\langle x + y\rangle_i = ([\![x + y]\!]_i, [\![m(x + y)]\!]_i)$.
- $\langle x + y\rangle \leftarrow \langle x\rangle + y$: Each cloud S_i takes $\langle x\rangle_i = ([\![x]\!]_i, [\![(m(x)]\!]_i)$ and its share of the MAC key α_i as the input. S_1 locally computes $[\![x + y]\!]_i = [\![x]\!]_i + y$, and for all other clouds S_i, simply sets $[\![x + y]\!]_i = [\![x]\!]_i$. All clouds (include S_1) compute $[\![m(x + y)]\!]_i = [\![m(x)]\!]_i + \alpha_i \cdot y$, and sets $\langle x + y\rangle_i = ([\![x + y]\!]_i, [\![m(x + y)]\!]_i)$.
- $\langle x \cdot y\rangle \leftarrow y \cdot \langle x\rangle$: Each cloud S_i takes $\langle x\rangle_i = ([\![x]\!]_i, [\![(m(x)]\!]_i)$ as the input, locally computes $[\![x \cdot y]\!]_i = y \cdot [\![x]\!]_i$ and $[\![m(x \cdot y)]\!]_i = y \cdot [\![m(x)]\!]_i$, then sets $\langle x \cdot y\rangle_i = ([\![x \cdot y]\!]_i, [\![m(x \cdot y)]\!]_i)$.

All the above computation can be done locally without interactions among the clouds. Multiplication over shares require an additional ideal functionality $\mathcal{F}_{\text{Triple}}$ (Figure 8) to generate Beaver's triple [7], i.e. $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ such that $ab = c$. Then multiplication can be performed as the following:

- $\langle x \cdot y\rangle \leftarrow \langle x\rangle \cdot \langle y\rangle$: Each cloud S_i takes $\langle x\rangle_i = ([\![x]\!]_i, [\![(m(x)]\!]_i)$, $\langle y\rangle_i = ([\![y]\!]_i, [\![m(y)]\!]_i)$ as the input. They also call $\mathcal{F}_{\text{Triple}}$ to receive a Beaver's triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, then:

1. The clouds compute $\langle \epsilon \rangle \leftarrow \langle x \rangle - \langle a \rangle$, $\langle \rho \rangle \leftarrow \langle y \rangle - \langle b \rangle$.
2. Each cloud S_i broadcasts $[\![\epsilon]\!]_i$ and $[\![\rho]\!]_i$, then opens $\epsilon = \sum_{i=1}^{n} [\![\epsilon]\!]_i$ and $\rho = \sum_{i=1}^{n} [\![\rho]\!]_i$ and broadcasts them.
3. The clouds compute $\langle x \cdot y \rangle \leftarrow \langle c \rangle + \epsilon \cdot \langle b \rangle + \rho \cdot \langle a \rangle + \epsilon \cdot \rho$

Functionality $\mathcal{F}_{\text{Triple}}$

- **Triple**: On input (*Triple*, id_a, id_b, id_c) from all clouds, sample two random values $a, b \overset{R}{\leftarrow} Z_N$ and set $(\mathsf{Val}[id_a], \mathsf{Val}[id_b], \mathsf{Val}[id_c]) \overset{R}{\leftarrow} (a, b, a \cdot b)$.
- **Open**: On input (*Open*, id) from all clouds, if $id \notin \mathbf{Val}$, ignore the message; Otherwise, send $\mathsf{Val}[id]$ to the adversary, wait x from the adversary and outputs x to all clouds.

Fig. 8: The Ideal Functionality $\mathcal{F}_{\text{Triple}}$

When performing a multiplication, the clouds needs to open the shares of ϵ and ρ. Opening the two values will not reveal any information about the client's data and the cloud's secret keys encoded in x and y. This is because $\epsilon = x - a$ and $\rho = y - b$ where a, b are uniformly random values that are not known by the clouds. Therefore ϵ and ρ leak no information about x and y. The clouds need to keep all the opened values and their MACs, and return them to the client along with the computation result. The client will check the MACs of opened value as well as the result, to ensure the computation was done correctly.

The ideal functionality $\mathcal{F}_{\text{Triple}}$ can be implemented using Overdrive [28] or MASCOT [27] running among the clouds. Or, if the client is honest and is trusted by all clouds, the client can generate the triples, since it has the MAC key.

6 Security Proof

Theorem 2. *Assuming factorization is hard and Paillier encryption scheme is CPA-secure, the ASSP protocols securely realizes $\mathcal{F}_{\text{ASSP}}$ in the presence of an adaptive, malicious PPT adversary in the $\mathcal{F}_{zk}^{R}, \mathcal{F}_{\text{Triple}}$-Hybrid Model.*

Proof. Throughout the proof, there are n clouds $S_1, \ldots, S_n$ and a client C. In the simulation, we construct a simulator $\mathcal{S}$ that invokes an adversary $\mathcal{A}$, who decides to corrupt a set of parties at the beginning, and later during the execution performs adaptive corruption by sending a message *corrupt* to the party it wants to corrupt. A set H contains all the honest parties. All parties not in H are corrupted. For convenience, we denote the set of all corrupted parties by I. The simulator knows both H and I at any point of time. To the adversary $\mathcal{A}$, the simulator plays the roles of the honest parties, until they are corrupted. The adversary controls the corrupted parties, and obtains the internal states when the party is corrupted. For all honest parties, unless required by the protocol, they erase messages received in the protocol after use.

Simulating Setup

The simulator $\mathcal{S}$ maintains a storage **sVal**. This storage will be used to store the simulated values such as shares of $\mathsf{crt}(0, s)$ for corrupted clouds. There are two cases depending on whether the client is corrupted or not.

- **Case 1**: Client has not yet been corrupted.
  - (a) $\mathcal{S}$ generates the Paillier key pair as the honest client will do, broadcasts the public key. It also simulates the ideal functionality $\mathcal{F}_{zk}^{R_N}$ for corrupted parties and sends $(\mathbf{proof}, 1)$ to each corrupted party. Then it sends (setup, p) to $\mathcal{F}_{\text{ASSP}}$.
  - (b) $\mathcal{S}$ then engages with each corrupted cloud in the cloud side parameters generation step. It encrypts the correct values in c_1, c_2 and c_4 as in the protocol, broadcasts them. It simulates $\mathcal{F}_{zk}^{R_R}$, $\mathcal{F}_{zk}^{R_p}$, $\mathcal{F}_{zk}^{R_M}$ for each corrupted party and sends $(\mathbf{proof}, 1)$ to the corrupted party. It also simulates $\mathcal{F}_{zk}^{R_S}$ for each corrupted party S_i, who is now the prover, and receives the witness $(\llbracket x \rrbracket_i, r_i)$ as the input from the corrupted party to the ideal functionality. $\mathcal{S}$ then checks whether the witness is correct and $\llbracket s \rrbracket_i$ is the correct discrete logarithm associated with Q_i. If not, sends *Abort* to $\mathcal{F}_{\text{ASSP}}$ and terminates, otherwise sends $(\mathit{Setup}, \llbracket s \rrbracket_i, Q_i)$ to $\mathcal{F}_{\text{ASSP}}$, and stores $\llbracket s \rrbracket_i$ in **sVal**. It also computes the value $-r_i$ as $\llbracket \mathsf{crt}(0, s) \rrbracket_i$ in **sVal**.
  - (c) For each honest cloud S_i (if any), $\mathcal{S}$ simply chooses a random number r_i and sets $\llbracket \mathsf{crt}(0, s) \rrbracket_i = -r_i$, then stores it in **sVal**. If all clouds have been corrupted, $\mathcal{S}$ must have already obtained the secret key s (from simulating $\mathcal{F}_{zk}^{R_S}$ and from K that is from $\mathcal{F}_{\text{ASSP}}$ when the last honest cloud was corrupted). Then it computes $\llbracket \mathsf{crt}(0, s) \rrbracket_0$ correctly as in the protocol, and stores it in **sVal**. If not all clouds have been corrupted, $\mathcal{S}$ chooses a random number $r_0 \in Z_N$ and sets $\llbracket \mathsf{crt}(0, s) \rrbracket_0 = r_0$ for now, then stores it in **sVal** – in this case the value will be updated once the shared cryptocurrency key are known by the simulator.
  - (d) Then after receiving r'_i from all corrupted clouds, $\mathcal{S}$ chooses r'_i for all $S_i \in H$, broadcasts them, then simulates the step of sharing α by broadcasting r_α computed as in the protocol. It also records $\llbracket \alpha \rrbracket_i$ for all clouds in **sVal** such that $\llbracket \alpha \rrbracket_1 = r'_1 + r_\alpha$ and $\llbracket \alpha \rrbracket_i = r'_i$ for the rest of the S_i.
  - (e) If during any step above, $\mathcal{A}$ decides to corrupt cloud S_i, $\mathcal{S}$ sends $(\mathit{corrupt}, i)$ to $\mathcal{F}_{\text{ASSP}}$ and gives S_i's state at the point to $\mathcal{A}$. The state of the corrupted cloud in the setup phase is $(c_1, c_2, \llbracket \alpha \rrbracket_i, \llbracket \mathsf{crt}(0, s) \rrbracket_i)$, part or full depending on the current progress of the simulation. In the tuple, c_1, c_2 are generated by $\mathcal{S}$, and $\llbracket \mathsf{crt}(0, s) \rrbracket_i, \llbracket \alpha \rrbracket_i$ are in **sVal**.
    If the party being corrupted is the Client, $\mathcal{S}$ sends $(\mathit{corrupt}, 0)$ to $\mathcal{F}_{\text{ASSP}}$. $\mathcal{S}$ needs to simulate its state. The Paillier private key and the MAC key (p, q, α) are generated by $\mathcal{S}$, and $\llbracket \mathsf{crt}(0, s) \rrbracket_0$ is in **sVal** (if needed). Then $\mathcal{S}$ continues the simulation following the strategy in Case 2 for subsequent simulation.
  - (f) If no one aborts during setup, $\mathcal{S}$ sends $(\mathit{Setup}, \mathit{go})$ for each honest party to $\mathcal{F}_{\text{ASSP}}$.
- **Case 2**: Client has already been corrupted and controlled by $\mathcal{A}$.

(a) In the client key generation step, $\mathcal{S}$ plays the roles of honest clouds. It receives the public key N, then simulates $\mathcal{F}_{zk}^{R_N}$ and receives $(\mathbf{prove}, N, p, q)$ from the corrupted client as its input to $\mathcal{F}_{zk}^{R_N}$. $\mathcal{S}$ checks to ensure the values are correct, otherwise sends *Abort* to $\mathcal{F}_{\text{ASSP}}$ and terminates. If not aborting, $\mathcal{S}$ sends $(Setup, p)$ to $\mathcal{F}_{\text{ASSP}}$.

(b) Next, in the cloud side parameters generation step, $\mathcal{S}$ receives c_1, c_2, c_4 from the corrupted client, simulates $\mathcal{F}_{zk}^{R_R}$, $\mathcal{F}_{zk}^{R_p}$, $\mathcal{F}_{zk}^{R_M}$, from which receives witness x_1, x_2 and verifies if (1) $x_1 = \mathsf{Dec}_{sk}(c_1) \wedge x_2 = \mathsf{Dec}_{sk}(c_2) \wedge 0 = \mathsf{Dec}_{sk}(c_4)$ and (2) $1 < x_1 < N$ and $1 < x_2 < N$, (3) $x_1 x_2 = 0$ and (4) $x_1 + x_2 = 1$. If yes, $\mathcal{S}$ puts c_1, c_2 in **sVal**. If not, $\mathcal{S}$ sends *Abort* to $\mathcal{F}_{\text{ASSP}}$ and terminates.

(c) $\mathcal{S}$ then simulates the messages from honest clouds to the client. For each cloud $S_i \in H$, $\mathcal{S}$ selects a random share $[\![s]\!]_i^*$ and r_i^*, computes $\hat{c}_i = (c_1 \boxdot [\![s]\!]_i^*) \boxplus \mathsf{Enc}_{pk}(r_i^*)$, returns $\hat{c}_i$ to $\mathcal{A}$. In addition, $\mathcal{S}$ simulates functionality $\mathcal{F}_{zk}^{R_S}$ by sending $(\mathbf{proof}, 1)$ to the corrupted client. Then $\mathcal{S}$ records $[\![s]\!]_i^*, r_i^*$ and $[\![\mathsf{crt}(0,s)]\!]_i = -r_i^*$ in **sVal**.

(d) To share the MAC key, $\mathcal{S}$ also chooses a random r_i' for each cloud $S_i \in H$, broadcasts it and receives r_α. It stores $[\![\alpha]\!]_i$ in **sVal** such that if $i = 1$ then $[\![\alpha]\!]_1 = r_i' + r_\alpha$, otherwise $[\![\alpha]\!]_1 = r_i'$.

(e) If during any step above, $\mathcal{A}$ decides to corrupt an honest cloud S_i, $\mathcal{S}$ sends $(corrupt, i)$ to $\mathcal{F}_{\text{ASSP}}$, receives $[\![s]\!]_i$, then needs to give S_i's state to $\mathcal{A}$. The state of the corrupted cloud in the setup phase is $(c_1, c_2, [\![\alpha]\!]_i, [\![\mathsf{crt}(0,s)]\!]_i)$, part or full depending on the current progress of the simulation. $\mathcal{S}$ has (c_1, c_2), and $[\![\alpha]\!]_i$. If $[\![\mathsf{crt}(0,s)]\!]_i$ is needed, $\mathcal{S}$ must also have $([\![s]\!]_i^*, r_i^*)$ in **sVal**. $\mathcal{S}$ now has the secret key share $[\![s]\!]_i$, and can find $r_i = \mathsf{crt}(0, [\![s]\!]_i^*) + r_i^* - \mathsf{crt}(0, [\![s]\!]_i)$. Then it computes $[\![\mathsf{crt}(0,s)]\!]_i = -r_i$, replaces the old value in **sVal** with it and includes it in the state to be sent to $\mathcal{A}$.

(f) If no one aborts during setup, $\mathcal{S}$ sends $(Setup, go)$ for each honest party to $\mathcal{F}_{\text{ASSP}}$.

Simulating Share

$\mathcal{S}$ maintains a data store **Value**. Each element in **Value** is of the form (id, v) such that id is an identifier and v is a value. $\mathcal{S}$ uses **Value** to track id being used for storing data in Val (in $\mathcal{F}_{\text{ASSP}}$), and also the value v being stored in $\mathsf{Val}[id]$. If $\mathcal{S}$ sees an id but does not know $\mathsf{Val}[id]$, $\mathcal{S}$ puts (id, null) in **Value**. $\mathcal{S}$ can update this item later when $\mathsf{Val}[id]$ is known. $\mathcal{S}$ also maintains a table **Shares**. Each row in **Shares** is for an id, and there are n columns, each for a cloud. The cell $\mathbf{Shares}_{(id,i)}$ stores $\langle e \rangle_i = ([\![e]\!]_i, [\![m(e)]\!]_i)$, which is the simulated authenticated share of S_i and $e = \mathsf{crt}(v, s)$ for the v linked to id, or null. Again, there are two cases depending on whether the client has been corrupted.

- **Case 1**: Client has not yet been corrupted.

(a) $\mathcal{S}$ takes a random d as the honest client's input. Then it chooses r_i, r_i' at random for the honest clouds (if any) and broadcasts them. $\mathcal{S}$ takes $[\![\mathsf{crt}(0,s)]\!]_0$

from **sVal** and sets $d' = \mathsf{crt}(d, 0) + [\![\mathsf{crt}(0, s)]\!]_0$. It then computes the partial MAC correctly as in the protocol. It receives all r_i, r'_i from the corrupted clouds and chooses r_i, r'_i for each honest clouds, then computes r_e, r_m as in the protocol and broadcasts them. $\mathcal{S}$ then sends $(share, id)$ for the corrupted clouds to $\mathcal{F}_{\text{ASSP}}$. It puts (id, d) in **Value**, and sets each $\mathbf{Share}_{(id,i)}$ with the shares as in the protocol. The shares $[\![e]\!]_i$ can be computed using r_i, r_e from the above, and $[\![\mathsf{crt}(0, s)]\!]_i$ in **sVal**. The shares $[\![m(e)]\!]_i$ can be computed using r'_i, r_m.

(*b*) Whenever $\mathcal{A}$ decides to corrupt an honest cloud S_i, $\mathcal{S}$ sends $(corrupt, i)$ to $\mathcal{F}_{\text{ASSP}}$.

i. If S_i is already the last honest cloud, $\mathcal{S}$ will get the secret key s from $\mathcal{F}_{\text{ASSP}}$. It also gets Val and knows the true value for each id. It then computes the correct $[\![\mathsf{crt}(0, s)]\!]_0$, and takes $[\![\mathsf{crt}(0, s)]\!]'_0$ that is the value currently in **sVal** (a random number selected when simulating setup). It replaces the old value in **sVal** with $[\![\mathsf{crt}(0, s)]\!]_0$. It also sets $[\![\mathsf{crt}(0, s)]\!]_i$ in **sVal** to $[\![\mathsf{crt}(0, s)]\!]_i - [\![\mathsf{crt}(0, s)]\!]'_0 + [\![\mathsf{crt}(0, s)]\!]_0$. For each id, it does the following: Take (id, d) from **Value**, and also $\mathsf{Val}(id)$. Compute the values $\mathsf{crt}(d, 0)$ and $\mathsf{crt}(\mathsf{Val}(id), 0)$. Take $\langle x \rangle_i = \mathbf{Share}_{(id,i)} = ([\![x]\!]_i, [\![m(e)]\!]_i)$, and compute $[\![x]\!]'_i = [\![x]\!]_i - \mathsf{crt}(d, 0) + \mathsf{crt}(\mathsf{Val}(id), 0) - [\![\mathsf{crt}(0, s)]\!]'_0 + [\![\mathsf{crt}(0, s)]\!]_0$, and $[\![m(x)]\!]'_i = [\![m(x)]\!]_i - \alpha \cdot \mathsf{crt}(d, 0) + \alpha \cdot \mathsf{crt}(\mathsf{Val}(id), 0)$. Then $\mathcal{S}$ sets the share $\mathbf{Share}_{(id,i)} = ([\![x]\!]'_i, [\![m(x)]\!]'_i)$.

ii. If there are still honest clouds beside the one being corrupted, $\mathcal{S}$ does not need to modify anything.

$\mathcal{S}$ then takes the i-th column from **Share** as the shares stored on S_i, $\mathcal{S}$ sends the shares along with S_i's state (all values can be found in **sVal**) to $\mathcal{A}$.

(*c*) When $\mathcal{A}$ decides to corrupt the client, $\mathcal{S}$ sends $(corrupt, 0)$ to $\mathcal{F}_{\text{ASSP}}$. Then it gives the state of the client to the adversary. The Paillier private key and the MAC key (p, q, α) are generated by $\mathcal{S}$, and $[\![\mathsf{crt}(0, s)]\!]_0$ is in **sVal**.

- **Case 2**: The client has already been corrupted.

(*a*) In Share protocol, the clouds have no input into the protocol. The only thing an honest cloud needs to do is to send random numbers r_i, r'_i. $\mathcal{S}$ simply chooses the random numbers for each honest cloud and broadcasts them as in the protocol. It also receives r_e, r_m from the corrupted client. Then it sends $(share, id)$ for each honest cloud to $\mathcal{F}_{\text{ASSP}}$, puts (id, null) in **Value** and for each id, sets $\mathbf{Share}_{(id,i)}$ with the shares as in the protocol. The shares $[\![e]\!]_i$ can be computed using r_i, r_e from above, and $[\![\mathsf{crt}(0, s)]\!]_i$ in **sVal**. The shares $[\![m(e)]\!]_i$ can be computed using r'_i, r_m. Note here $\mathbf{Share}_{(id,j)}$ for all corrupted clouds S_j are set to null and they are not needed.

(*b*) Whenever $\mathcal{A}$ corrupts cloud S_i, if there is at least one honest cloud besides the one being corrupted, $\mathcal{S}$ sends $(corrupt, i)$ to $\mathcal{F}_{\text{ASSP}}$, receives $[\![s]\!]_i$. If $([\![s]\!]^*_i, r^*_i)$ exists in **sVal**, compute the correct $[\![\mathsf{crt}(0, s)]\!]_i$ using them and $[\![s]\!]_i$, then replace the old value with the new one. It also modifies the shares in the i-th column of **Share** with $[\![\mathsf{crt}(0, s)]\!]_i$. It hands S_i's state and shares to $\mathcal{A}$. All values in the state can be found in **sVal**.

If S_i being corrupted is the last honest cloud, then $\mathcal{S}$ sends $(corrupt, i)$ to $\mathcal{F}_{\text{ASSP}}$ and receives Val and $s = \sum_{i=1}^{n} K[i]$ from $\mathcal{F}_{\text{ASSP}}$. $\mathcal{S}$ then creates the correct state for S_i. In the state $c_1, c_2, [\![\alpha]\!]_i$ are in **sVal** and can be used without modification. However, $[\![\mathsf{crt}(0,s)]\!]_i$ needs to be made consistent. $\mathcal{S}$ also needs to make the shares holds by S_i consistent.

i. If the client was corrupted after $[\![\mathsf{crt}(0,s)]\!]$ had been generated, then the share $[\![\mathsf{crt}(0,s)]\!]_0$, now is revealed, is a random number r_0 chosen by $\mathcal{S}$ in the simulation. To ensure consistency, i.e. $\sum_{i=0}^{n}[\![\mathsf{crt}(0,s)]\!]_i = \mathsf{crt}(0,s)$, $\mathcal{S}$ computes $\mathsf{crt}(0,s)$. $\mathcal{S}$ can do so because it knows secret key s and p, q. Then it sets $[\![\mathsf{crt}(0,s)]\!]_i = -r_0 + \mathsf{crt}(0,s)$ (all the other shares besides $[\![\mathsf{crt}(0,s)]\!]_i$ and $[\![\mathsf{crt}(0,s)]\!]_0$ are correct). Then $\mathcal{S}$ takes $[\![\mathsf{crt}(0,s)]\!]'_i$ which is the current value in **sVal** and replaces it with $[\![\mathsf{crt}(0,s)]\!]_i$. For each id, if (id, null) is in **Value**, $\mathcal{S}$ sets the share to $\mathbf{Share}_{(id,i)} = \mathbf{Share}_{(id,i)} - [\![\mathsf{crt}(0,s)]\!]'_i + [\![\mathsf{crt}(0,s)]\!]_i$; if (id, d) is in **Value** for some $d \neq \mathsf{null}$, $\mathcal{S}$ also takes $\mathsf{Val}[id]$ and modifies the share as in Case 1, (b).

ii. If the client was corrupted before $[\![\mathsf{crt}(0,s)]\!]$ had been generated, then $\mathcal{S}$ has $([\![s]\!]^*_i, r^*_i)$ in **sVal**. It computes the correct $[\![\mathsf{crt}(0,s)]\!]_i$ using them and the obtained secret key s. It then takes $[\![\mathsf{crt}(0,s)]\!]'_i$ that is the current value in **sVal**, and replaces the old value with the new one. For all id, $\mathcal{S}$ sets each share $\mathbf{Share}_{(id,i)} = \mathbf{Share}_{(id,i)} - [\![\mathsf{crt}(0,s)]\!]'_i + [\![\mathsf{crt}(0,s)]\!]_i$.

Simulating Reconstruct

Again there are two cases depending on whether the client has been corrupted:

- **Case 1**: Client has not yet been corrupted.
  - (a) $\mathcal{S}$ receives a ciphertext from each corrupted cloud S_i, and recovers the authenticated share $\langle x \rangle_i$.
    - i. If not all clouds have been corrupted, $\mathcal{S}$ checks if the received shares are correct, by comparing each received share with the share stored in **Share**. If any share is not correct, then $\mathcal{S}$ sends $(Abort)$ to $\mathcal{F}_{\text{ASSP}}$ and terminates. Otherwise $\mathcal{S}$ sends out $(Reconstruct, id)$ to $\mathcal{F}_{\text{ASSP}}$.
    - ii. If all clouds have been corrupted, $\mathcal{S}$ checks whether the MAC is valid, if not then $\mathcal{S}$ sends $(Abort)$ to $\mathcal{F}_{\text{ASSP}}$ and terminates. Otherwise it sends $(reconstruct, id)$ to $\mathcal{F}_{\text{ASSP}}$, then recovers the value from shares sent by the corrupted clouds, then sends it to $\mathcal{F}_{\text{ASSP}}$.
  - (b) Corruption is handled in the same way as in simulating the Share protocol, case 1 (b) and (c).
- **Case 2**: Client has already been corrupted.
  - (a) $\mathcal{S}$ checks if id is in **Value**, if not ignores the message, otherwise $\mathcal{S}$ gets the share for each honest cloud from **Shares**, computes the ciphertext as in the protocol and returns the ciphertext and the MAC share to the adversary. Note if all clouds have been corrupted, since the client are corrupted as well, the adversary will no longer interact with the simulator and the simulation is trivial.

(b) Corruption is handled in the same way as in simulating share, case 2 (*b*).

Simulating computation

With regard to computation, it is among the clouds without involving the client. If all clouds are corrupted, then it is trivial to simulate. If there are still honest clouds, the simulator $\mathcal{S}$ simulates as follows:

- $\langle x + y\rangle, \langle x\rangle + y, y \cdot \langle x\rangle$: They are performed locally, the simulator simply sends instructions with the correct *id* and values to the ideal functionality.
- $\langle x \cdot y\rangle$: The simulator $\mathcal{S}$ simulates $\mathcal{F}_{\text{Triple}}$, then sends ϵ and ρ to adversary $\mathcal{A}$ and waits for it to input ϵ' and ρ'. If $(\epsilon, \rho) \neq (\epsilon', \rho')$, send *Abort* to $\mathcal{F}_{\text{ASSP}}$. Then compute locally with the shares in **Share**, and instruct $\mathcal{F}_{\text{ASSP}}$ with the correct *id* and values.

Corruption is handled as in simulating the Share protocol, case 1 (*b*) and (*c*), or case 2 (*b*) depending on whether the client is corrupted.

7 Experimental Evaluation

We have implemented a prototype of ASSP in C++. The source code of the implementation is available online[1]. We used OpenSSL[2] (version 1.0.2g) for the multi-precision integer operations and the underlying cryptographic primitives. The parties were implemented as different processes communicating through network (remote procedure calls). We used gRPC[3] (version 0.14) for network communication. We used multi-threading to parallelize the communication. Each party runs l threads where l is the number of all other parties. Each thread handles the communication to and from another party. Each party also has another thread for computation.

All the experiments were carried out in Amazon AWS Cloud. The client and the clouds were running on different on-demand instances. The client was running on a C5.2xlarge instance in the Oregon (US West) data center. The instance has 8 vCPUs (4 physical cores) based on the Intel Xeon Platinum 8000 series (Skylake-SP) CPUs, 32 GB RAM and an up to 10Gbps LAN connection. To simulate multiple cloud providers, we deployed the cloud parties at different data centers in the US: Ohio (US East), N. Virginia (US East), and Northern California (US West). The cloud parties were running on R5.2xlarge instances that have 8 vCPUs (4 physical cores) based also on the Intel Xeon Platinum 8000 series CPUs, 64 GB RAM, and an up to 10Gbps LAN connection. When experimenting with more than 3 clouds, we co-located no more than two parties in the same data center (on different instances).

We first report the performance of the Setup protocol. In Table 1 we show the average running time and communication cost based on 100 runs for each

[1] Url is removed for blind review.
[2] `https://www.openssl.org/`
[3] `https://www.grpc.io/`

# clouds	**—N—**	**Running Time (s)**		**Comm.**
		Parallel Comm.	**Serial Comm.**	**(KB)**
2	2048	1.98	2.76	93.02
	3072	8.72	10.01	122.12
3	2048	2.89	3.36	139.64
	3072	11.40	13.60	185.12
4	2048	3.95	4.15	186.47
	3072	18.00	20.20	244.35
5	2048	4.44	5.84	252.89
	3072	18.30	23.20	305.96
6	2048	6.31	6.88	282.15
	3072	23.90	25.50	367.20

Table 1: Performance of the Setup Protocol (running time excludes the Paillier key generation time)

set of parameters. In the experiment we varied the number of clouds from 2 to 6, and tested with 2048-bit and 3072-bit N. The running time showed in Table 1 does not include the time for generating the Paillier key pair. This is because the time for Paillier key pair generation is much larger than all other time and varies significantly across different runs (because the algorithm is probabilistic). For key pairs based on 2048-bit N, the time ranged from 9.56 - 17.42 seconds; for 3072-bit N, the time ranged from 13.12 - 104.57 seconds. Note that even though the key pair generation requires significant time, the Setup protocol only needs to be run once. Therefore, it is not a major performance concern. In Table 1, we show two sets of running time: the first set was measured with multi-threading to parallelize the communication, and the second set, as a comparison, was measured with a single thread that handles all communications in a serial manner. As we can see from Table 1, the running time increases in the number of clouds. This is mainly because the client runs ZKPoK with clouds pair-wisely. We can also see that parallelizing communication does improve the overall performance. For the ease of implementation, We only used one thread for the computation. In principle, multi-threading can be implemented to parallelize the computation, and it would be effective in reducing the overall running time. The communication cost was measured at the client side, and is the total amount of incoming and outgoing data. As we can see, the communication cost is small.

Next, we show the performance of the Share and Reconstruct protocols. In the experiment, we tested with 6 clouds. We varied the number of data items being shared and reconstructed from 10 to $100,000$ (10^5). We measured the total running time of the protocol (from the client sending the request to all parties completing the work) and communication cost (total amount of incoming and outgoing data measured at the client side). As we can see, the Share protocol is fast. The time for sharing 10^5 data items is only 31.4 seconds when $|N| = 2048$ and 49.1 seconds when $|N| = 3072$. The Reconstruct protocol is slower. This is because a Paillier decryption is required for each item being reconstructed. The decryption time dominates the total running time of the Reconstruct protocol. For example, when $|N| = 3072$, the total time for reconstructing 10^5 data el-

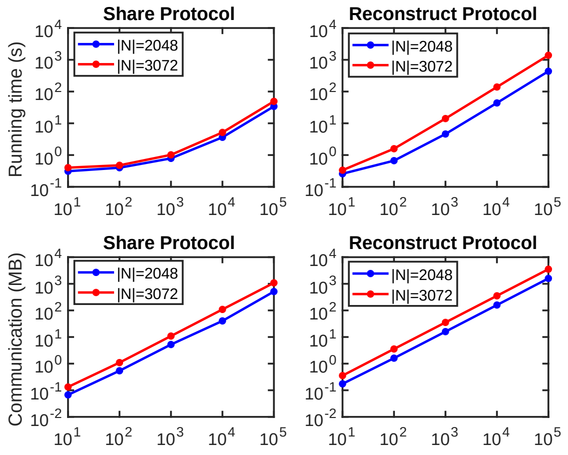

Fig. 9: Performance of the Share and Reconstruct protocols (logarithmic scale)

ements was 1385.73 seconds, and decryption took 1267.32 seconds. Since each decryption is independent, multi-threading could be used to reduce the total running time. We can also see from the figure that the communication cost of the protocols is linear in the number of element (slope ≈ 1 in the logarithmic scale plot).

# Mult	Share Size	2 Clouds	4 Clouds	6 Clouds
10	2048	0.081	0.083	0.083
	3072	0.16	0.16	0.16
10^2	2048	0.40	0.41	0.39
	3072	0.38	0.41	0.41
10^3	2048	0.92	0.90	0.93
	3072	0.91	0.96	0.97
10^4	2048	1.65	1.73	1.88
	3072	2.14	2.27	2.25
10^5	2048	10.16	10.35	10.50
	3072	13.84	13.38	13.97

Table 2: Running time (seconds) of MPC multiplication

We then show the performance of MPC based on ASSP. Here we show the performance of share multiplication, because other operations involve only local computation and are much faster (see Section 5.6). In the experiment, we used 2048 or 3072 bit N, 2 , 4 and 6 clouds, and varies the number of multiplications from 10 to 10^5. In the experiment, we used the code in the SPDZ-2 repository[4] for the computation. We used the SPDZ compiler [29] that takes an program written in Python and executes it in MPC over the SPDZ online protocol. We used the same programs but with different sizes of shares in ASSP as the input, and measured the running time of the online phase computation. As we can see, the performance of ASSP based MPC is reasonable. The maximum throughput of ASSP is about 10,000 multiplication per second when using 2048-bit shares and 7,000 when using 3072-bit shares. There is a difference in speed when using shares

[4] `https://github.com/bristolcrypto/SPDZ-2`

with different sizes. As expected, shorter shares lead to better MPC performance. The result suggests that to improve the performance of ASSP based MPC, we should investigate how to make the shares shorter in the future.

8 Related Work

Secret Sharing. There have been abundant work aiming to enhance secret sharing in one way or another. Multi-secret sharing aims to use the same set of shares for multiple secrets [20, 10, 24, 14]. Password protected secret sharing [6, 25, 26, 44] combines secret sharing and user authentication. The user can retrieve the secret from n servers if he has the correct password and there are at least $t+1$ honest servers. The shared data remains secret against an adversary who corrupts at most t servers. Note that in password protected secret sharing, the data itself is not shared but rather encrypted under a threshold encryption scheme. This means the client cannot directly use the data for outsourced MPC. Leakage resilient secret sharing [32, 8, 40] requires that the secrecy of shared data is maintained in the presence of an adversary who may learn bounded information about *all* the shares. The aim of leakage resilient secret sharing is to prevent an adversary from learning the secret through side-channels, rather than through accessing the shares directly. Therefore it is not suitable for our senario.
Self-enforcing penalties. The concept of self-enforcing penalties was firstly proposed in [19] to prevent illegal redistribution of digital content. The idea is to encrypt the content so that it can be decrypted using a key of the same length as the content itself, or a short key that contains some sensitive information like credit card number of the user. The assumption (in 1996), was that bandwidth was a scarce resource that would prevent the user from disclosing the long key, thus the user who illegally redistributing the content would be penalized by revealing the sensitive information to others. More recently, there have been a line of work on leveraging cryptocurrency to self-enforce direct financial penalties in the event of confidentiality breaches. Kiayias and Tang [30] proposed leakage-deterring public key primitives. Essentially, the public key is embedded with some owner-specific private data (e.g. a Bitcoin private key), so that if a users shares an unauthorized device (or implementation) for private key functionalities, e.g. decryption or signing, the private data can be extracted by those who use the devices. Similarly, the same authors [31] proposed traitor deterring scheme (TDS), a multi-recipient public key encryption scheme in which an authority issues secret keys to a set of users after the users provide their bitcoin secret keys as a form of collateral. The traitor deterring property ensures that if a malicious coalition of traitors produces an unauthorized decryption device, any one who can use the device will be able to recover at least one of the traitors' bitcoin secret keys by accessing the device in a blackbox fashion. The core idea in [30, 31] is to correlate the bitcoin key with the secret key (for decryption or signing) and embed the correlation in the public key, then allow query the decryption or signing oracle in a certain way to find the correlation, then the bitcoin key can be extracted from the public key with the knowledge of

the correlation. This idea however cannot be applied to secret sharing because in secret sharing, there is no keys nor cryptographic functions that can be used as oracles. Another self-enforcing scheme is by Mangipudi et. al. [35], in which a user and a publisher run a protocol such that the user receives a multimedia file that has a digital watermark encoding a bitcoin key from the user. The publisher generates the watermark obliviously and knows nothing about the bitcoin key. However, if the user shares the file with the public, the publisher, after getting the shared file, can extract the bitcoin key from the watermark. Note this scheme only allows the publisher who generated the watermark originally to extract the bitcoin key. Also digital watermarking generally is only applicable to multimedia data that is not sensible to small changes.

9 Conclusion and Future Work

In this paper, we presented ASSP with its application to secure cloud computing. ASSP is an authenticated secret sharing scheme with shares being constructed on top of the ring Z_N. The ring allows us to pack the data from a client and some secret key from clouds together, to enforce financial penalties automatically if the client's data in the clouds is compromised. ASSP is MPC friendly, so that the client can securely outsource computation by instructing the cloud servers to perform MPC directly on the shares. The share authentication feature, through a partial MAC scheme, further ensures the integrity of the outsourced data and the correctness of the computation results. We formally proved the security of ASSP in the presence of a malicious, adaptive adversary. We also showed through experimental evaluation that the performance of ASSP is reasonable.

In the future, we would like to further improve the efficiency of ASSP by reducing the share size. Currently, the security of ASSP relies on the factorization problem. This leads to the large N, and in consequence, large shares. We would like to investigate alternative constructions that can be based on smaller rings. Another future direction is on the deposit mechanisms. Currently, the cloud has to pay a deposit for each contract. This would prevent the cloud from having many clients at the same time because individual deposits will accumulate into an unaffordable amount. We would like to find a more scalable deposit mechanism.

References

1. Bitcoin. `https://bitcoin.org/`
2. Bitcoin transaction. `https://en.bitcoin.it/wiki/Transaction`
3. Ethereum. `https://www.ethereum.org/`
4. Zcash. `https://z.cash/`
5. Archer, D.W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J.I., Smart, N.P., Wright, R.N.: From keys to databases - real-world applications of secure multi-party computation. Comput. J. **61**(12), 1749–1771 (2018)
6. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: ACM CCS. pp. 433–444. ACM (2011)

7. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO. pp. 420–432. Springer (1991)
8. Benhamouda, F., Degwekar, A., Ishai, Y., Rabin, T.: On the local leakage resilience of linear secret sharing schemes. In: CRYPTO. pp. 531–561 (2018)
9. Bing, C., Stubbs, J., Menn, J.: China hacked hpe, ibm and then attacked clients. `https://www.reuters.com/article/us-china-cyber-hpe-ibm-exclusive/exclusive-china-hacked-hpe-ibm-and-then-attacked-clients-sources-idUSKCN1OJ2OY` (2018)
10. Blundo, C., Santis, A.D., Crescenzo, G.D., Gaggia, A.G., Vaccaro, U.: Multi-secret sharing schemes. In: CRYPTO (1994)
11. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS
12. Camenisch, J., Michels, M.: Proving in zero-knowledge that a number is the product of two safe primes. In: EUROCRYPT. pp. 107–122. Springer (1999)
13. Cramer, R., Fehr, S., Ishai, Y., Kushilevitz, E.: Efficient multi-party computation over rings. In: EUROCRYPT. pp. 596–613. Springer (2003)
14. Crescenzo, G.D.: Sharing one secret vs. sharing many secrets. Theor. Comput. Sci. **295**, 123–140 (2003)
15. Damgård, I., Jurik, M., Nielsen, J.B.: A generalization of paillier's public-key system with applications to electronic voting. International Journal of Information Security **9**(6), 371–385 (2010)
16. Damgård, I., Orlandi, C., Simkin, M.: Yet another compiler for active security or: efficient mpc over arbitrary rings. In: CRYPTO. Springer (2018)
17. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO, pp. 643–662. Springer (2012)
18. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: ACM CCS. pp. 211–227. ACM (2017)
19. Dwork, C., Lotspiech, J., Naor, M.: Digital signets: Self-enforcing protection of digital information. In: STOC. vol. 28, pp. 489–498 (1996)
20. Franklin, M., Yung, M.: Communication complexity of secure computation (extended abstract). In: STOC. pp. 699–710 (1992)
21. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ecdsa with fast trustless setup. In: ACM CCS. pp. 1179–1194. ACM (2018)
22. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
23. Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols - Techniques and Constructions. Springer (2010)
24. Jackson, W.A., Martin, K., O'Keefe, C.: Ideal secret sharing schemes with multiple secrets. Journal of Cryptology **9**(4), 233–250 (1996)
25. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and t-pake in the password-only model. In: ASIACRYPT. pp. 233–253. Springer (2014)
26. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: Toppss: Cost-minimal password-protected secret sharing based on threshold oprf. In: ACNS. pp. 39–58. Springer (2017)
27. Keller, M., Orsini, E., Scholl, P.: Mascot: faster malicious arithmetic secure computation with oblivious transfer. In: ACM CCS. pp. 830–842. ACM (2016)
28. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT. pp. 158–189 (2018)
29. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: ACM CCS. pp. 549–560 (2013)

30. Kiayias, A., Tang, Q.: How to keep a secret: leakage deterring public-key cryptosystems. In: ACM CCS. pp. 943–954. ACM (2013)
31. Kiayias, A., Tang, Q.: Traitor deterring schemes: Using bitcoin as collateral for digital content. In: ACM CCS. pp. 231–242. ACM (2015)
32. Kumar, A., Meka, R., Sahai, A.: Leakage-resilient secret sharing. Tech. rep., Cryptology ePrint Archive, Report 2018/1138, 2018. https://eprint. iacr. org
33. Lindell, Y., Nof, A.: Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody. In: ACM CCS. pp. 1837–1854. ACM (2018)
34. Lipmaa, H.: Statistical zero-knowledge proofs from diophantine equations. IACR Cryptology ePrint Archive **2001**, 86 (2001)
35. Mangipudi, E.V., Rao, K., Clark, J., Kate, A.: Automated penalization of data breaches using crypto-augmented smart contracts. IACR Cryptology ePrint Archive **2018**, 1050 (2018)
36. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. pp. 223–238. Springer (1999)
37. Poupard, G., Stern, J.: Short proofs of knowledge for factoring. In: PKC. pp. 147–166. Springer (2000)
38. Rivest, R.L.: All-or-nothing encryption and the package transform. In: FSE. pp. 210–218 (1997)
39. Shoup, V.: A computational introduction to number theory and algebra. Cambridge university press (2009)
40. Srinivasan, A., Vasudevan, P.N.: Leakage resilient secret sharing and applications. Tech. rep., Cryptology ePrint Archive, Report 2018/1154, 2018. https://eprint. iacr. org (2018)
41. Takabi, H., JosHi, J., aHn, G.J.: Security and privacy challenges in cloud computing environments. IEEE Security Privacy **8**(6), 24–31 (2010)
42. Todd, P.: OP_CHECKLOCKTIMEVERIFY. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki` (2014)
43. Trappler, T.: In the cloud, a data breach is only as bad as your contract. `https://www.computerworld.com/article/2501695/in-the-cloud--a-data-breach-is-only-as-bad-as-your-contract.html` (2012)
44. Yi, X., Tari, Z., Hao, F., Chen, L., Liu, J.K., Yang, X., Lam, K.Y., Khalil, I., Zomaya, A.Y.: Efficient threshold password-authenticated secret sharing protocols for cloud computing. Journal of Parallel and Distributed Computing (2019)