Authenticating compromisable storage systems

Yu, Jiangshan; Ryan, Mark; Chen, Liqun

DOI:
10.1109/Trustcom/BigDataSE/ICESS.2017.216

Document Version
Peer reviewed version

Citation for published version (Harvard):
Yu, J, Ryan, M & Chen, L 2017, Authenticating compromisable storage systems. in *Proceedings of 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-17).* IEEE Computer Society Press, 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-17), Sydney, Australia, 1/08/17. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.216

Authenticating compromisable storage systems

Jiangshan Yu
Interdisciplinary Center for Security, Reliability and Trust
University of Luxembourg
Email: jiangshan.yu@uni.lu

Mark Ryan
School of Computer Science
University of Birmingham
Email: m.d.ryan@cs.bham.ac.uk

Liqun Chen
Department of Computer Science
University of Surrey
Email: liqun.chen@surrey.ac.uk

***Abstract*—A service may be implemented over several servers, and those servers may become compromised by an attacker, e.g. through software vulnerabilities. When this happens, the service manager will remove the vulnerabilities and re-instate the server. Typically, this will involve regenerating the public key by which clients authenticate the service, and revoking the old one.**

This paper presents a scheme which allows a storage service composed of several servers to create a group public key in a decentralised manner, and maintain its security even when such compromises take place. By maintaining keys for a long term, we reduce the reliance on public-key certification. The storage servers periodically update the decryption secrets corresponding to a public key, in such a way that secrets gained by an attacker are rendered useless after an update takes place. An attacker would have to compromise all the servers within a short period lying between two updates in order to fully compromise the system.

***Index Terms*—Post compromise security, Proactive security, Self-healing system, Authentication, Information security.**

I. Introduction

a) Motivation: Services across the internet authenticate themselves to clients by means of public keys. If a server becomes compromised by an attacker, the attacker may obtain the secret key corresponding to its public key. If this happens, in a real situation, the service manager will eventually become aware of it, and can then shut down the service, repair the damage, and generate a new key pair. The service manager also has to revoke the old public key. In this scenario, clients of this server will need to switch from the old revoked public key to the newly updated key.

This paper introduces a technique which allows a service composed of several servers to create a a service level public key in a decentralised manner in a way that tolerates compromise of the secret key. Specifically, even if the servers that run the service are compromised and the attacker obtains their secrets, the service owner is (in situations satisfying our assumptions) able to re-establish the servers with new secrets, and securely run the service with *the same public key* as before.

By allowing public keys to survive possible compromises of the services that own them, the technique reduces the reliance on certificate authorities [1], and avoids the complexities and uncertainties of public key revocation. Data storage systems [2], [3], industrial control systems [4], and corporate servers [5] whose clients are other corporate servers are examples of situations in which one could benefit from having long-lived public keys.

We develop these ideas in the context of an enterprise backup service. We assume a platform which offers a storage service provided collectively by a set of independent servers. Each server generates its private/public key pair without requiring recourse to a trusted dealer. A client Alice selects a group of servers, and requests that they collaborate using the protocol of this paper to compute a public key (which we call the "group public key"). The system provides features such that

- Alice can always authenticate the service (and encrypt her backup[1]) by using the fixed group public key that she obtained securely in the initialisation;
- Alice is not required to maintain any decryption secret for later data recovery;
- in the event of a disaster that destroys her local data, Alice can recover the encrypted backup by providing her identity to each server;
- Alice can be guaranteed that the outsourced encrypted backup remains secure even if an attacker compromises all the servers over a long time period.

b) Overview of our solution: To apply the technique, we assume that the storage service is composed of several servers. Each server creates a key pair independently. The servers collectively run a protocol to compute a group public key based on the public part of their individual key pairs. We call the secret part of each server's individual key pair a "share" of the decryption secret w.r.t. their group public key. The full decryption secret is never reconstructed in the protocol.

Time is divided into epochs. At each epoch, the servers engage in a protocol to proactively update their individual key pairs in a way that connects them back to the group public key. In particular, (A) the group public key remains unchanged regardless to the update process ensuring that it can still be authenticated; (B) data encrypted using the fixed group public key can still be decrypted by using the new secrets; and (C) each update renders useless all previously generated shares of the decryption secrets. So, an attacker cannot recover an encrypted data without compromising all servers' shares of the decryption secrets in the same epoch.

[1]In practice, Alice can use hybrid encryption: she encrypts her backup with a fresh symmetric key, and that symmetric key becomes the data encrypted with the group public key. The symmetric key will be destroyed after the encryption, and Alice is not required to remember any secret.

At any time, one or more of the servers may become compromised by an attacker. In that case, the attacker obtains all the data (including secrets) stored on the server. Our solution ensures that the attacker is not able to perform any decryption, provided that in any epoch at least one of the servers remains uncompromised.

The solution we present requires all the selected servers to participate. At the end of the paper, we discuss how it can be extended to a threshold-based solution.

c) Contribution: We introduce a system which allows a family of servers to maintain decryption secrets corresponding to a group public key, with the following properties:

- The secrets are proactively maintained. This means that the servers periodically engage in a protocol to update the secrets, defending from attackers that could potentially compromise all the servers over a long period.
- The group public key remains fixed, even though the decryption secrets of the servers change in each time period. We achieve this in a simple and decentralised manner without requiring a trusted dealer.
- There is no time during our protocol at which an entire decryption key corresponding to the group public key is generated, stored, reconstructed or used.

We present the system as a self-healing storage service, though this construction can be used as a building block to solve the problem of authenticating a compromisable service in other applications. Self-healing means servers can be attacked and their secrets compromised, but after attacks the service can continue with the same public key as before.

We formalise an adversary model for this kind of system. Since there might be robust malware that cannot be removed from a server, our adversary model allows the adversary to permanently compromise servers. We also define the security of a distributed storage system against our model through a security game. We provide a rigorous formal security proof of the proposed system under the defined security model. Our proof also shows that the proposed scheme provides IND-CCA2 security. Due to space limitations, the security proof is not presented in this paper. We refer readers to our technical report [6] for a fully detailed formal security analysis.

d) Efficiency: The system is also optimal in round communication between a client and a group of servers, i.e. it requires only one round communication per-server in both phases for data encryption/distribution and for data reconstruction, and does not require any client involvement for the periodic update. In addition, it requires only two exponentiation operations on the client side for encryption or decryption.

II. Security model

This section first presents an informal attacker model and security goal, in Section II-A and Section II-B, respectively. It then defines the formal security model in Section II-C.

We consider the scenario that an attacker wants to steal the sensitive data of users on cloud servers, by gradually breaking into servers of the system.

TABLE I
THE EXPLANATION ON DIFFERENT TYPES OF PARTICIPANTS.

Notation	Description
$\mathcal{S}_{PAC}$	The set of servers that are permanently controlled by attackers. Security actions, e.g. software patches and malware removal, can not succeed in restoring the servers to a secure state.
$\mathcal{S}_{TAC}$	The set of servers that are temporarily controlled by attackers. Security actions, e.g. software patches and malware removal, can succeed in restoring the servers to a secure state
$\mathcal{S}_{Sec}$	The set of servers that are currently secure.
$\mathcal{C}_{AC}$	The set of clients (i.e. data owners) that are controlled by attackers.
$\mathcal{C}_{Sec}$	The set of clients that are currently secure.
$\mathcal{S}_{\text{Alice}}$	The set of servers selected by client Alice.
$\mathcal{S}$	The complete set of all servers, such that $\mathcal{S} = \mathcal{S}_{PAC} \cup \mathcal{S}_{TAC} \cup \mathcal{S}_{Sec}$
$\mathcal{C}$	The complete set of all clients, such that $\mathcal{C} = \mathcal{C}_{AC} \cup \mathcal{C}_{Sec}$
$\mathcal{P}$	The complete set of all participants, such that $\mathcal{P} = \mathcal{S} \cup \mathcal{C}$.

A. Attacker model

Suppose an attacker compromises a server. Then the attacker can fully control the server and has access to all its stored secrets. Suppose sometime later, the maintainer of the server applies software patches and malware removal. Depending on the nature of the compromise, that action might restore the server into a secure state, or it might not.

As shown in Table I, we use $\mathcal{S}_{PAC}$ to represent the set of permanently attacker-controlled servers; $\mathcal{S}_{TAC}$ to represent the set of temporarily attacker-controlled servers (as illustrated in Figure 2); and $\mathcal{S}_{Sec}$ to represent the set of secure servers.

Figure 1 shows the possible transformation between different types of servers. Generally, any secure server in $\mathcal{S}_{Sec}$ may become a temporarily attacker-controlled server; any server in $\mathcal{S}_{TAC}$ may become a secure server; and any server in $\mathcal{S}_{Sec}$ or $\mathcal{S}_{TAC}$ may become a permanently attacker-controlled server.

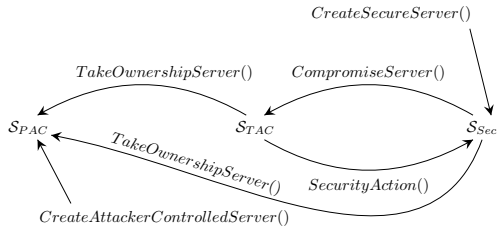

Fig. 1. A figure presenting the possible transformation between different types of servers. In our formal security model, these transformations can be achieved by using oracle queries as defined in Section II-C.

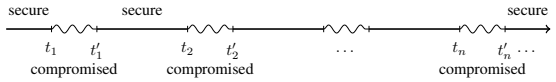

Fig. 2. A timeline presenting a server's security state transformation between a temporarily attacker controlled server S_{TAC} and a secure server S_{Sec}. For all $i > 0$, we assume that the server is compromised in the time interval between t_i and t'_i, and is secure in the time interval between t'_i and t_{i+1}.

B. Security goal

All servers update their secrets simultaneously at predetermined times. We say T is an epoch if T starts from the beginning of the process for updating secrets, and ends at the beginning of the next process for updating secrets. Note that since we allow an adversary to corrupt servers at any moment during an epoch, if a server is corrupted during an update phase from epoch T to the next epoch T', we consider the attacker being able to obtain secrets in both the T-th and T'-th epochs.

Let $\mathcal{S}_{\text{Alice}}$ be the set of servers selected by Alice. At a given epoch T, let $\mathcal{S}_{PAC}(T)$ be the set of permanently attacker-controlled servers in $\mathcal{S}_{\text{Alice}}$, and $\mathcal{S}_{TAC}(T)$ the set of temporarily attacker-controlled servers in $\mathcal{S}_{\text{Alice}}$.

Our security goal is that an attacker cannot learn any secret of Alice, provided the total number of attacker-controlled servers in T and T' is less than the number of servers chosen by Alice, i.e. $|\mathcal{S}_{PAC}(T')| + |\mathcal{S}_{TAC}(T)| + |\mathcal{S}_{TAC}(T')| < |\mathcal{S}_{\text{Alice}}|$.

Remark 1: Loosely speaking, it says that the system should be secure if the total number of compromised servers in any epoch is less than the number of servers chosen by Alice. Note that $\mathcal{S}_{PAC}(T)$ is the set of permanently compromised servers at epoch T, and these servers will be included in the set of permanently compromised servers in future epochs as well. So we have $|\mathcal{S}_{PAC}(T)| \leq |\mathcal{S}_{PAC}(T')|$. However, this is not true for $\mathcal{S}_{TAC}(\cdot)$.

C. Formal Model

We first define the scenario we are considering, i.e. attackers can periodically compromise cloud servers for storage. Then we formally define the ability of an attacker, and the security of a self-healing distributed storage system.

Definition 1: A **periodically compromised system environment (PCSE)** is an environment in which an attacker can periodically control participants of a protocol. It consists of

1) Protocol Π: the underlying security protocol;
2) Security checking oracle $SecurityCheck(\Pi, S)$: given a server $S \in \mathcal{S}$ in protocol Π, it outputs a value V_S to indicate if S is compromised. If $V_S = comp$, then an attacker has compromised S; otherwise, S is secure. This models the security status of a server.
3) Security action oracle $SecurityAction(\Pi, S)$: given a server S, it outputs a strategy for S such that if S is a temporarily attacker-controlled server, i.e. $S \in \mathcal{S}_{TAC}$, and it executes the strategy, then the server will become a secure server, i.e. $SecurityCheck(\Pi, S) = secure$.

We define our security model through a game with two participants, namely a challenger and a probabilistic polynomial time (PPT) adversary $\mathcal{A}$. The attacker's goal is to win the game that is initialised by the challenger. $\mathcal{A}$ is able to ask the following oracle queries.

1) $\mathcal{O}_1$: $Settings(\Pi)$. By sending this query, $\mathcal{A}$ is given all the public parameters of Π.
2) $\mathcal{O}_2$: $Execute(\Pi, \mathcal{P}')$. Upon receiving this query, the set of participants $\mathcal{P}' \subseteq \mathcal{P}$ executes protocol Π, if applicable (where $\mathcal{P}$ is the set of all participants, as defined in Table I). The exchanged messages will be recorded and sent to $\mathcal{A}$. This oracle query models an attacker's ability to eavesdrop communications between participants in Π.
3) $\mathcal{O}_3$: $CreateAttackerControlledClient(\Pi, C)$. Upon receiving this query with a fresh identity C, the oracle creates an attacker-controlled client C in Π according to the attacker's choice. After this query has been made, we have that $\mathcal{C}_{AC} := \mathcal{C}_{AC} \cup \{C\}$. We say an identity is "fresh" if and only if the identity is unique and has not been previously generated. This oracle models an attacker's ability to register a new client of its choice.
4) $\mathcal{O}_4$: $CreateAttackerControlledServer(\Pi, S)$. Upon receiving this query, the oracle creates a fresh server S, and sends the corresponding secret key and public key to the attacker. (The created secret key will be used as this server's share of the group decryption secret.) After this query has been made, we have that $\mathcal{S}_{PAC} := \mathcal{S}_{PAC} \cup \{S\}$. This oracle allows $\mathcal{A}$ to adaptively register permanently attacker-controlled servers of its choice.
5) $\mathcal{O}_5$: $CreateSecureClient(\Pi, C)$. Upon receiving this query, the oracle creates a fresh client C in Π. After this query has been made, we have that $\mathcal{C}_{Sec} := \mathcal{C}_{Sec} \cup \{C\}$. This oracle query allows an attacker to introduce more clients, which are initially secure.
6) $\mathcal{O}_6$: $CreateSecureServer(\Pi, S)$. Upon receiving this query, the oracle creates a fresh server S in Π. After this query has been made, we have that $\mathcal{S}_{Sec} := \mathcal{S}_{Sec} \cup \{S\}$. This oracle query allows an attacker to introduce more servers, which are initially secure.
7) $\mathcal{O}_7$: $CompromiseClient(\Pi, C)$. Upon receiving this query for some $C \in \mathcal{C}_{Sec}$ in Π, the oracle forwards all corresponding secrets of C to $\mathcal{A}$. From now on the attacker controls C so that $C \in \mathcal{C}_{AC}$ and $C \notin C_{Sec}$ after this query has been made. This oracle query allows $\mathcal{A}$ to adaptively and permanently compromise a client of its choice.
8) $\mathcal{O}_8$: $TakeOwnershipServer(\Pi, S)$. Upon receiving this query for some $S \in \mathcal{S}_{Sec}$ or $S \in \mathcal{S}_{TAC}$ in Π, the oracle forwards all corresponding secrets of S to $\mathcal{A}$, and from now on the attacker controls S. So, S is moved from its current set in to $\mathcal{S}_{PAC}$ after this query has been made. This oracle query allows $\mathcal{A}$ to adaptively and *permanently* compromise a server of its choice.

9) $\mathcal{O}_9$: $CompromiseServer(\Pi, S)$. Upon receiving this query, the oracle outputs all secrets of $S \in \mathcal{S}_{Sec}$ in Π. We have $S \notin S_{Sec}$ and $S \in S_{TAC}$ after this query has been made. This oracle query models $\mathcal{A}$'s ability to adaptively and *temporarily* compromise an attacker-controlled server of its choice.
10) $\mathcal{O}_{10}$: $Dec(\Pi, Enc(M, PK_{\mathcal{S}_C}), C)$. Upon receiving this query for some client $C \in \mathcal{C}_{Sec}$ with data M, the set $\mathcal{S}_C$ of servers collectively executes the decryption protocol to decrypt the encrypted data $Enc(M, PK_{\mathcal{S}_C})$, and sends the decryption result M to the attacker, where $PK_{\mathcal{S}_C}$ is the group public key of the set $\mathcal{S}_C$ servers selected by C for encryption/decryption.

We now consider the distributed storage scenario. If a powerful attacker $\mathcal{A}$ can fully control a data owner's device when the device is creating or recovering data M, then $\mathcal{A}$ can easily learn M. As mentioned before, we do not consider this case as there is nothing we can do and it is not interesting. To focus on the more interesting cases, we only consider that $\mathcal{A}$ cannot learn M by compromising the data owner's device during the secret creation or recovery time.

Definition 2: A self-healing distributed storage protocol Π comprises a group of data owners, and a group $\mathcal{S}$ of decryptors. It consists of two algorithm and three protocols, namely key generation algorithm $KeyGen(\cdot)$, encryption algorithm $Enc(\cdot)$, encryption key construction protocol Π_{PK}, decryption key construction protocol Π_{SK}, and decryption protocol Π_{dec}.

- $KeyGen(\lambda)$. Taking security parameter λ as input, it outputs a pair (s_i, P_i) of private and public keys for the decryptor $S_i \in \mathcal{S}$.
- Π_{PK}. It is run by a group $\mathcal{S}' \subseteq \mathcal{S}$ of decryptors. The decryptor S_i has private input s_i. After the completion of the protocol, each $S_i \in \mathcal{S}'$ outputs the same long term public key $PK_{S'}$ of group $\mathcal{S}'$.
- Π_{SK}. It is run by a group $\mathcal{S}' \subseteq \mathcal{S}$ of decryptors at each time period j. After the completion of the protocol, each $S_i \in \mathcal{S}'$ outputs a share s_{ij} of the corresponding group private decryption key for time period j. If $j = 0$, then the private input of S_i is s_i. If $j > 0$, then the input of S_i is $s_{i(j-1)}$.
- $Enc(M, PK_{S'})$. Taking data M and the public key $PK_{S'}$ of group $\mathcal{S}'$ as input, it outputs the ciphertext of M encrypted under $PK_{S'}$.
- Π_{dec}. It is run by a data owner with encrypted M, and a group $\mathcal{S}' \subseteq \mathcal{S}$ of decryptors with their private input s_{ij} for each $S_i \in \mathcal{S}'$ at time period j. After the completion of the protocol, the data owner outputs M.

Remark 2: The self-healing feature is defined by the *decryption key construction* protocol Π_{SK}, as it enables decryptors to update their keys periodically.

Definition 3: A self-healing distributed storage protocol Π is (k, n)-*secure* if the advantage $Adv_{\mathcal{A},n,k}(\lambda) = |Pr[b = b'] - \frac{1}{2}|$ of $\mathcal{A}$ to win the following game, denoted *Game-PCSE*, is negligible in the security parameter λ.

Game-PCSE:

- $Setup(\Pi, \lambda)$. The challenger sets up protocol Π according to the security parameter λ. Initially, $\mathcal{S} = \mathcal{C} = \emptyset$.
- Query phase. The attacker can ask a polynomially bounded number of oracle queries $\mathcal{O}_i$ for $i \in \{1, 2, \ldots, 10\}$. Let j_4, j_8, and j_9 be counters counting the total number of $\mathcal{O}_4$, $\mathcal{O}_8$, and $\mathcal{O}_9$ queries asked by the attacker, respectively. We have that $j_4 + j_8 + j_9 < k$.
- Security action phase. The challenger makes security checking oracle queries on all servers, and then makes security action oracle queries on the servers that are temporarily controlled by the attacker. At the end of this phase, the counter j_9 will be reset to "0".
- The query phase and the security action phase are repeated a polynomially bounded number of times.
- $Challenge(C_b, C)$. The attacker selects a target client C who has not been asked through $\mathcal{O}_i$ for $i \in \{3, 7\}$, i.e. $C \in \mathcal{C}_{Sec}$; and selects two messages M_0 and M_1, s.t. $|M_0| = |M_1|$. The attacker then sends them to the challenger. The challenger tosses a coin. Let $b \in \{0, 1\}$ be the result of the coin toss. The challenger then encrypts M_b according to Π, and sends the ciphertext $C_b = Enc(M_b, PK_{\mathcal{S}})$ back to the attacker.
- The query phase and the security action phase are repeated a polynomially bounded number of times. Additionally, we require that the target client C cannot be asked through $\mathcal{O}_3$ and $\mathcal{O}_7$, and $Dec(\Pi, C_b, C)$ cannot be queried through $\mathcal{O}_{10}$.
- $Guess(b)$. The attacker makes a guess b' of the value of b, and outputs b'. The attacker wins if $b = b'$.

Remark 3: In the game defined above, the execution of a query phase followed by a security action phase simulates an epoch of the protocol.

Remark 4: In a (k, n)-threshold cryptosystem, an attacker can break the security if the attacker is able to compromise k secrets/parties during the lifetime of the system. However, in the above defined (k, n)-secure system in the PCSE, an attacker cannot break the security even if the attacker can compromise all n parties in the lifetime of the system, provided at any time point t between two updates, at most $k-1$ parties are compromised by the attacker.

III. Our solution

We present our solution, first in a non-threshold form (i.e., we stipulate that the minimum number k of servers needed for performing decryption is equal to n, the total number of servers). Later, in section VI-A, we generalise it to a threshold-based solution where we allow one to choose $k < n$.

A. Basic idea

A client Alice selects a set of servers, and requests that they collaborate using the Initialisation protocol (detailed later) to compute a public key for a storage service. She encrypts her sensitive data using this group public key. Each selected server stores a copy of the encrypted data. (Of course, in practice, this makes sense only if the data is small. If a large amount of

Setup

$S_A : (a, g^a)$, $S_B : (b, g^b)$, $S_C : (c, g^c)$, $PK = g^{abc}$

Zero-th Update

$S_A : (a_0, g^{a_0})$, $S_B : (b_0, g^{b_0})$, $S_C : (c_0, g^{c_0})$, $H_0 = g^{(a_0 b_0 c_0 / abc)}$

First Update

$S_A : (a_1, g^{a_1})$, $S_B : (b_1, g^{b_1})$, $S_C : (c_1, g^{c_1})$, $H_1 = g^{(a_1 b_1 c_1 / abc)}$

The j-th Update

$S_A : (a_j, g^{a_j})$, $S_B : (b_j, g^{b_j})$, $S_C : (c_j, g^{c_j})$, $H_j = g^{(a_j b_j c_j / abc)}$

Encryption at any time

$C = (\alpha = g^{abck}, \beta = MZ^k)$, for some data M and random number k

Decryption at the j-th epoch

Compute $\gamma = e(\alpha, H_j)$, then decrypt (β, γ) by using (a_j, b_j, c_j)

Fig. 3. The data associated with the servers S_A, S_B and S_C at different stages of the protocol, and the encryption and decryption computations.

data is required to be stored, we assume that a fresh session key is generated. The large data is encrypted with the session key and stored on an (untrusted) cloud service; and the session key is encrypted using the group public key and stored on the selected servers.)

Time is divided into epochs. At the end of each epoch, the servers execute a protocol during which they generate new decryption keys and destroy the old ones.

If a server is compromised in an epoch, the attacker obtains all its (shares of) decryption keys. However, the protocol ensures that decryption keys from a server in one epoch cannot be used together with decryption keys from a server in a different epoch. Each change of epoch renders useless the decryption keys obtained by the attacker in previous epochs.

Thus, to decrypt the secret, an attacker would have to compromise a threshold number of servers *within the same* epoch.

B. Abstract construction

Our protocol is based on bilinear map, as defined below.

Definition 4 (Bilinear Map): Let G_1, G_2 be two cyclic groups of a sufficiently large prime order q. A map $e : G_1^2 \to G_2$ is said to be bilinear if $e(g^a, g^b) = e(g, g)^{ab}$ is efficiently computable for all $g \in G_1$ and $a, b \in \mathbb{Z}_q$; and e is non-degenerate, i.e. $e(g, g) \neq 1$.

We now explain the protocol with three servers, S_A, S_B, and S_C. Let $g \in G_1$ be a fixed public value and $Z = e(g, g) \in G_2$. The data associated with the servers at different stages of the protocol are presented in Fig. 3.

Setup and zero-th epoch. S_A generates a private key a, and a public key g^a. Similarly, S_B and S_C generate (b, g^b) and (c, g^c). Then S_A, S_B, S_C collectively compute and publish their joint public key g^{abc}.

Next, S_A generates a new key a_0 and public key g^{a_0}, and similarly S_B and S_C generate (b_0, g^{b_0}) and (c_0, g^{c_0}). Then S_A, S_B, S_C collectively compute and publish helper data $H_0 = g^{(a_0/a)\cdot(b_0/b)\cdot(c_0/c)}$ with proofs that they have correctly performed the computation. They destroy the secrets a, b, c.

At the end of the $(j-1)$-th epoch. The servers replace their decryption keys a_{j-1}, b_{j-1}, and c_{j-1} with new ones a_j, b_j, and c_j. Then S_A, S_B, S_C collectively compute and publish helper data $H_j = g^{(a_j/a)\cdot(b_j/b)\cdot(c_j/c)}$ with proofs that they have correctly performed the computation. The values a, b, c are not required to compute H_j. They destroy the secrets a_{j-1}, b_{j-1}, c_{j-1}.

Encryption of data M. At any time during the server lifecycle (i.e. any epoch j), a client Alice can encrypt her data M by using the (unchanging) public key g^{abc}. She selects a new random k, and computes $C = (\alpha = g^{abck}, \beta = MZ^k)$.

Decryption of ciphertext (α, β) at the j-th epoch. After authenticating client Alice's request for decryption, the servers can collectively decrypt a ciphertext (α, β) during any epoch. To decrypt (α, β), the servers compute $\gamma = e(\alpha, H_j) = Z^{a_j b_j c_j k}$. Then the servers use their secrets a_j, b_j, and c_j to collectively compute Z^k, and then they can recover the data M from $\beta = MZ^k$. Note that during this decryption process, Alice should apply masking to the γ to prevent servers from learning the plaintext. More details are presented in the next section.

Remark 5: Note that the public key used by clients for encryption remains constant regardless of the secret updates on the server side. Also, the update procedure is independent of the number of stored ciphertexts. That is because in the update phase the servers need only collectively compute the helper data. The ciphertext (α, β) of each data item remains unchanged.

C. Detailed construction

Initialisation: Setup. Let G_1, G_2 be two groups of a sufficiently large prime order q, such that $|q| = \lambda$, with a bilinear map $e : G_1^2 \to G_2$, and $g \in G_1$ is a random generator and $Z = e(g, g) \in G_2$.

Let $S_1, S_2, \ldots, S_n$ be the servers selected by Alice. S_i performs $KeyGen(\lambda)$ to create setting-up key pair (s_i, g^{s_i}), for some $s_i \in \mathbb{Z}_q$, respectively. They also run Π_{PK} to compute a group public key $PK = g^{\prod s_i}$, which is available to the client in an authentic matter, e.g., via a certificate. This key can be established as follows:

- Each S_i computes and publishes $P_i = g^{s_i}$.
- S_2 computes $PK_{12} = (P_1)^{s_2}$. This computation can be verified by other servers by checking $e(PK_{12}, g) = e(P_1, P_2)$.
- S_i computes $PK_{1\ldots i} = (PK_{1\ldots(i-1)})^{s_i}$, which again can be verified by other servers by checking $e(PK_{1\ldots i}, g) = e(PK_{1\ldots(i-1)}, P_i)$.

Now, each S_i has a secret key s_i and a group public key PK.

Zero-th epoch. This epoch is to generate shares of the first decryption key through Π_{SK}. Each S_i chooses another secret s_{i0}, computes $u_{i0} = s_{i0}/s_i$, computes and publishes $P_{i0} = g^{s_{i0}}$, $P'_{i0} = g^{u_{i0}}$ and deletes s_i. The correctness of these values can be checked as $e(P'_{i0}, P_i) = e(P_{i0}, g)$.

By using u_{i0}, S_i works with other servers to get $H_0 = g^{\prod s_{i0}/s_i}$ in the same way as computing PK, and then deletes u_{i0}.

At the end of the initialisation, S_i only holds its share s_{i0} at secret. This value can be used for decryption (if needed) and is used for the next decryption key update. In addition, S_i also holds two public values, namely a helper data H_0 and a group public key PK.

Note that the group public key is used for data encryption by the clients. This implies that the clients do not have to follow the server key updating processes, and they will keep using the key PK for a reasonably long time.

Updating the decryption keys (Π_{SK}): The decryption key update process is similar to the computation of the first decryption keys presented in the previous phase. At the end of the $(j-1)$-th epoch for some $j \geq 1$, the servers replace their decryption keys $s_{i(j-1)}$ with new ones, s_{ij}. This is achieved as follows.

- With the input $s_{i(j-1)}$, S_i chooses s_{ij}, computes $u_{ij} = s_{ij}/s_{i(j-1)}$, computes and publishes $P_{ij} = g^{s_{ij}}$ and $P'_{ij} = g^{u_{ij}}$, and deletes $s_{i(j-1)}$. The correctness of these values can be checked as $e(P'_{ij}, P_{i(j-1)}) = e(P_{ij}, g)$.
- By using u_{ij}, S_i works with other servers to get $H_j = H_{j-1}^{\prod s_{ij}/s_{i(j-1)}} = g^{\prod s_{ij}/s_i}$ and then deletes u_{ij}. The correctness of these values should also be verified in the same way as computing PK.

At the end of $(j-1)$-th update, S_i only holds s_{ij}. This value is used for both decryption and update in the (j)-th epoch.

Encryption: To encrypt data M, Alice selects a new random k, and computes $PK^k = g^{k \cdot \prod s_i}$ and MZ^k. Alice sends $(\alpha = PK^k, \beta = MZ^k)$ to each server.

Servers only accept (α, β) as some encrypted data from Alice if a valid proof of knowledge of M (or k) is provided. This is used to prevent replay attacks in which an attacker who has observed (α, β) sets up an account with the servers, and provides (α, β) as the attacker's encrypted data, then requests servers to decrypt it for the attacker. Any secure zero knowledge proof of knowledge (ZKPK) can be used. For example, the proof can be a Schnorr ZKPK of k, where the prover knows k and the verifier knows PK^k. If the prover shows knowledge of k, this implies that she also knows M.

At the end, Alice destroys M and k after all servers are convinced and accepted the ciphertext.

Decryption (Π_{dec}): The basic idea of the decryption process is presented in Fig. 4.

In more detail, in the j-th epoch for some $j \geq 0$, Alice sends a request to a selected server for retrieving the encrypted data. After successfully authenticating Alice, the server calculates $\gamma = e(\alpha, H_j) = Z^{k \cdot \prod s_{ij}}$, and sends (β, γ) to Alice.

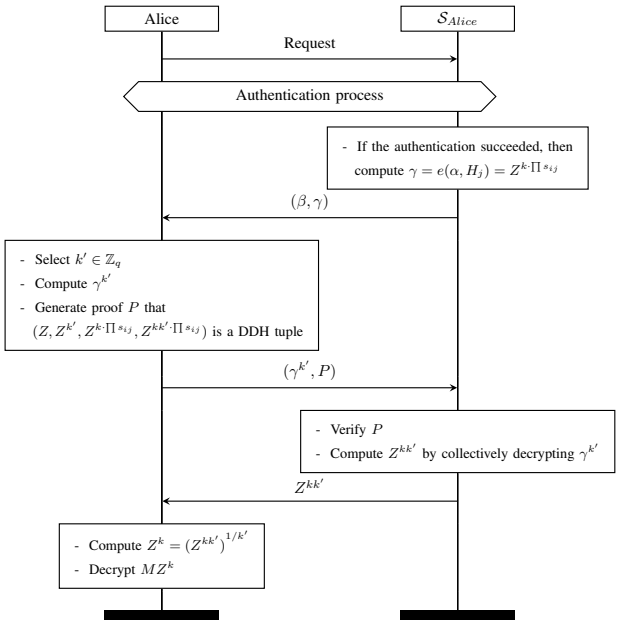

Fig. 4. The basic idea of the decryption process.

Alice selects a new random $k' \in \mathbb{Z}_q$, sends $Z^{k'}$ to each of the servers as her commitment on k', computes $\gamma^{k'} = Z^{kk' \cdot \prod s_{ij}}$, and asks each server to remove its s_{ij} from the exponent by calculating $\gamma^{k' \cdot s_{ij}^{-1}}$. The final output should be $Z^{kk'}$. She then can recover Z^k by computing $(Z^{kk'})^{1/k'}$, and thus be able to decrypt MZ^k.

Before a server decrypts some message requested by a user, the server expects a proof that the requested decryption is indeed a step to help the user to recover a key that the user actually owns, i.e. to prove that

$$(Z, Z^{k'}, Z^{k \cdot \prod s_{ij}}, Z^{kk' \cdot \prod s_{ij}})$$

is a DDH tuple. This can be done by using classic non-interactive ZKPK schemes, for proving that (g, g^x, g^y, g^{xy}) is a DDH tuple (e.g. Chaum-Pedersen protocol [7]). Each server also needs to check the received values from other servers in the same way.

IV. Security analysis

The full formal security proof can be found in our technical report [6] (Section 4, page 6-9). Here we provide a summary of our security analysis. Our goal is to prove the security of our protocol per Definition 3; that is, we prove that the advantage that a probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ has to win the *Game-PCSE* is negligible.

To provide a rigorous security proof, we formally define our hardness assumption, and define a cryptographic game accordingly. To better simulate our protocol, the defined game

TABLE II
A LIST OF NOTATIONS FOR EVALUATION

Notation	Description
Exp	the operation of modular exponentiation;
Mul	the operation of modular multiplication;
Inv	the operation of modular multiplicative inverse;
BP	the bilinear pairing operation $e(\cdot)$ of mapping from $G_1 \times G_1$ to G_2;

supports multiple rounds simulating the epochs in our protocol. We call such a game with j rounds as *j-round modified decisional bilinear Diffie-Hellman inversion* game, denoted *Game-j-R-MDBDHI*. We then prove in a lemma that if an adversary can win *Game-j-R-MDBDHI* with a non-negligible advantage, then we can break our hardness assumption.

Finally, we prove in our theorem that if an adversary can win *Game-PCSE* with a non-negligible advantage ϵ, then we can make use of this adversary to win *Game-j-R-MDBDHI* with advantage $\frac{(1+2\epsilon)(N-N'')(N-N')}{8N^2}$, which is also non-negligible. (Here, the values N, N' and N'' are quantities of servers participating in the protocol.)

V. PERFORMANCE EVALUATION

This section evaluates the performance of the proposed system in two aspects, namely the number of communication rounds and the computational cost. We define some notations to facilitate our evaluation, as presented in Table II.

The number in front of a notation means the number of times this computation is required, e.g., $3Mul$ means that Mul operation has been calculated three times in this phase. Note that since the initialisation phase will only be run once at the beginning of the system, our performance evaluation ignores it here. In addition, we assume that the zero knowledge proof uses Schnorr NIZK scheme [8], and costs $1Exp + 1Mul$ for proof generation and $2Exp + 1Mul$ for verification [9].

Table III presents the number of communication rounds in each phase in three figures. In brief, our evaluation shows that the protocol for updating the decryption keys, which will be run periodically, does not require the involvement of any client. In addition, only one communication round is needed for a client to communicate with each group member for data encryption/distribution and for data reconstruction. This results in N rounds for data encryption, and $N + 1$ rounds for data decryption, where N is the number of servers. The reason that the data decryption requires $N + 1$ rounds (rather than N rounds) is that apart from one communication round with each of the N servers, an additional communication round is required between the client and the first server that the client communicates with for computing γ, as shown in the Figure 4. However, as we will see later, although the number of communication rounds that a client is involved is dependent on the number of servers, a client's computational cost is independent of the number of servers.

Table IV presents the computational cost, for a client and for a server, in different protocol phases. Our evaluation shows that all bilinear pairing operations are done on the server side. Due to limited space, we refer readers to our full version [6] for a more detailed analysis.

TABLE III
THE NUMBER OF COMMUNICATION ROUNDS .

	Updating the decryption keys	Encryption	Decryption
Rounds involving client	-	N rounds	N+1 rounds
Rounds involving only servers	N-1 rounds	-	N-1 rounds
Rounds in total	N-1 rounds	N rounds	2N rounds

TABLE IV
THE COMPUTATIONAL COST IN DIFFERENT PROTOCOL PHASES .

Entity/phase	Updating the decryption keys	Encryption	Decryption
Client	-	3 Exp +2 Mul	4 Exp + 1 Mul + 1 Inv
Server	1 Inv + 3 Exp + 4(N-1) BP	2 Exp + 1 Mul	1 BP* + 3 Exp + 1 Mul + (N-1) BP**

Note:
* Only 1 out of N servers need to perform this operation (for computing γ with cost 1BP).
** The number of correctness checks is different for each server depending on their network position, where (N-1)BP is the cost in average.

VI. DISCUSSION AND RELATED WORK

A. Extension to a threshold system

As mentioned in previous sections, our system requires the presence of all servers for recovering a secret. However, it can be easily extended to a threshold-based system, by using any classical (verifiable) secret sharing schemes to back-up all ephemeral secret keys of servers.

To be more precise, let 'key servers' be the servers in our protocol and 'back-up servers' be the secret sharing servers. Each time a new key of a key server is generated, the key will be distributed to a set of back-up servers through secret sharing schemes, and the shares associated to the old keys will be destroyed. So, when a key server is dead, our system can still continue by recovering the dead server's secret keys from shares, and take actions from there to re-build the server.

Intuitively, the extended threshold system is secure even if we additionally allow an attacker to compromise less than a threshold number of back-up servers at any epoch, provided that the set of back-up servers does not overlap with the set of key servers, and all key servers uses the same threshold with the same set of back-up servers for sharing their keys. Loosely speaking, since the shares of different epochs are independent each other, the compromise of shares in an epoch does not help an attacker to recover secrets shared in other epochs.

In fact, we can easily improve the security guarantee of the extended threshold system by letting key servers use different

sets of back-up servers for sharing their keys. In this way, an attacker would need to compromise a threshold number of back-up servers to only obtain the secret of a single key server, rather than being able to recover all key servers' secrets. A more rigorous security analysis of the extended threshold system will be our future work.

B. Related work

Outsourcing storage is a growing industry, it enables users to remotely store their data into a cloud, reduces users' burden of in-house infrastructure maintenance, and offers economies of scale. However, due to concerns over data privacy and security [10], users are not willing to outsource their sensitive data in the cloud [11]. For example, many recent attacks have been perpetrated on cloud systems [12], [13]. We discuss related proactively secure systems that have been designed to secure against compromised servers.

Proactive secret sharing (PSS) (e.g., [14]–[18]) is a technique for sharing a secret among a set of servers; it is secure against an attacker that can compromise servers, one by one, over a long period. In PSS, as in our protocol, time is divided into epochs. In each epoch, the servers that hold shares of the secret engage in a protocol to update their shares. An attacker may compromise some servers in a given epoch, but the learnt secrets are useless in other epochs. Thus, even if all the servers are eventually compromised over different epochs, the secret remains intact provided that in each epoch there was at least one server that remained honest.

Proactively secure cryptographic systems (e.g., [19]–[25]) apply the ideas of proactively secure secret sharing to sharing decryption or signing secrets among several servers. Such systems have been achieved by combining a proactively secure secret sharing scheme with an encryption or signature scheme. However, these constructions make use of a trusted dealer, who creates the secret key and distributes shares of some secrets to the servers. Unfortunately, the creation of the secret key in a single location by the dealer prevents the decentralisation required and achieved in our protocol. Although it is mentioned in a number of papers (e.g. [25] can be extended to a dealer-less protocol by using [26]) that the function of the trusted dealer in these schemes can be done by the servers, it is well known that both distributing a secret in Shamir's secret sharing scheme and creating and distributing an RSA (or a DL) key, amongst multiple players without a trusted dealer, are complicated and very expensive.In this paper, we propose a decentralised distributed decryption scheme, which does not have such a trusted dealer and is efficient.

VII. Conclusion

Increasing numbers of attacks on cloud servers challenge the security of cloud storage. We have introduced a provably secure distributed storage system as a security enhanced approach to this challenge. Because the system updates decryption secrets on servers, it remains secure even if all the servers are compromised over a long time, provided that no more than a threshold number of servers are compromised in a single epoch. The storage system maintains a fixed public key; the key can be used securely even after such compromises. This solves an important problem of how to authenticate servers when they are compromisable, without having to rely on PKI.

References

[1] J. Clark and P. C. van Oorschot, "SSL and HTTPS:revisiting past challenges and evaluating certificate trust model enhancements," in *IEEE Symposium on Security and Privacy*, 2013.

[2] M. T. Khorshed, A. B. M. S. Ali, and S. A. Wasimi, "A survey on gaps, threat remediation challenges and some thoughts for proactive attack detection in cloud computing," *Future Generation Comp. Syst.*, vol. 28, no. 6, pp. 833–851, 2012.

[3] "The treacherous 12: Cloud computing top threats in 2016," Cloud Security Alliance Reports, February 2016.

[4] F. Skopik, G. Settanni, and R. Fiedler, "A problem shared is a problem halved: A survey on the dimensions of collective cyber defense through security information sharing," *Computers & Security*, vol. 60, pp. 154–176, 2016.

[5] K. Bode, "Google: Gmail now fully encrypted between data centers, servers," DSL Reports, March 2014.

[6] J. Yu, M. Ryan, and L. Chen, "Authenticating compromisable storage systems," Cryptology ePrint Archive, Report 2017/485, 2017, http://eprint.iacr.org/2017/485.

[7] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *CRYPTO*, 1992, pp. 89–105.

[8] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *CRYPTO*, 1989, pp. 239–252.

[9] F. Hao, "Schnorr NIZK Proof: Non-interactive Zero Knowledge Proof for Discrete Logarithm," Internet Draft 04, July 2016.

[10] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From security to assurance in the cloud: A survey," *ACM Comput. Surv.*, vol. 48, no. 1, p. 2, 2015.

[11] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *CCSW*, 2009, pp. 85–90.

[12] F. Pennic, "Anthem suffers the largest healthcare data breach to date," https://goo.gl/6npFbO, 2015.

[13] T. Greene, "Biggest data breaches of 2015," https://goo.gl/B9Oo2a, 2015.

[14] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks (extended abstract)," in *ACM PODC*, 1991, pp. 51–59.

[15] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO*, 1995, pp. 339–352.

[16] V. Nikov and S. Nikova, "On proactive secret sharing schemes," in *SAC 2004, Waterloo, Canada, August 9-10*, 2004, pp. 308–325.

[17] D. A. Schultz, B. Liskov, and M. Liskov, "MPSS: mobile proactive secret sharing," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 4, p. 34, 2010.

[18] J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky, "How to withstand mobile virus attacks, revisited," in *ACM PODC*, 2014, pp. 293–302.

[19] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung, "Proactive RSA," in *CRYPTO* , 1997, pp. 440–454.

[20] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung, "Optimal resilience proactive public-key cryptosystems," in *FOCS*, 1997, pp. 384–393.

[21] Y. Frankel, P. D. MacKenzie, and M. Yung, "Adaptively-secure optimal-resilience proactive RSA," in *ASIACRYPT*, 1999, pp. 180–194.

[22] Y. Frankel, P. MacKenzie, and M. Yung, "Adaptive security for the additive-sharing based proactive RSA," in *PKC*, 2001, pp. 240–263.

[23] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *ACM CCS*, 2002, pp. 88–97.

[24] J. F. Almansa, I. Damgård, and J. B. Nielsen, "Simplified threshold RSA with adaptive and proactive security," in *EUROCRYPT*, 2006, pp. 593–611.

[25] D. Boneh, X. Boyen, and S. Halevi, "Chosen ciphertext secure public key threshold encryption without random oracles," in *CT-RSA*, 2006, pp. 226–243.

[26] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *J. Cryptology*, vol. 20, no. 1, pp. 51–83, 2007.