Secure Key-Updating for Lazy Revocation

Michael Backes Christian Cachin Alina Oprea*

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{mbc,cca,opr}@zurich.ibm.com

September 25, 2005

Abstract

We consider the problem of efficient key management and user revocation in cryptographic file systems that allow shared access to files. A performance-efficient solution to user revocation in such systems is lazy revocation, a method that delays the re-encryption of a file until the next write to that file. We formalize the notion of key-updating schemes for lazy revocation, an abstraction to manage cryptographic keys in file systems with lazy revocation, and give a security definition for such schemes. We give two composition methods that combine two secure key-updating schemes into a new secure scheme that permits a larger number of user revocations. We prove the security of two slightly modified existing constructions and propose a novel binary tree construction that is also provable secure in our model. Finally, we give a systematic analysis of the computational and communication complexity of the three constructions and show that the novel construction improves the previously known constructions.

1 Introduction

The recent trend of storing large amounts of data on high-speed, dedicated storage-area networks (SANs) stimulates flexible methods for information sharing, but also raises new security concerns. As the networked storage devices are subject to attacks, protecting the confidentiality of stored data is highly desirable in such an environment. Several cryptographic file systems have been designed for this purpose [14, 26, 21, 15], but practical solutions for efficient key management and user revocation still need to be developed further.

We consider cryptographic file systems that allow shared access to stored information and that use untrusted storage devices. In such systems, we can aggregate files into sets such that access permissions and ownership are managed at the level of these sets. The users who have access to the files in a set form a group, managed by the owner of the files, or the *group owner*. Initially, the same cryptographic key can be used to encrypt all files in a set, but upon revocation of a user from the group, the key needs to be changed to prevent access of revoked users to the files. The group owner generates and distributes this new key to the users in the group. There are two options for handling user revocation: *active* and *lazy* revocation, which differ in the way that users are revoked from a group. With active revocation, all files in a set are immediately re-encrypted with the new encryption key. The amount of work caused by a single revocation with this method might, however, be prohibitive for large sets of files. With the alternative method of lazy revocation, re-encryption of a file is delayed until the next write to that file and, thus, users do not experience disruptions

*Permanent address: Computer Science Department, Carnegie Mellon University, Pittsburgh, USA. Email: alina@cs.cmu.edu

in the operation of the file system caused by the immediate re-encryption of all files protected by the same revoked key. In systems adopting lazy revocation, the files in a set might be encrypted with different keys. Storing and distributing these keys becomes more difficult than in systems using active revocation.

In this paper, we address the problem of efficient key management in cryptographic file systems with lazy revocation. An immediate solution to this problem, adopted by the first cryptographic file systems using delayed re-encryption [14], is to store all keys for the files in a set at the group owner. However, we are interested in more efficient methods, in which the number of stored keys is not proportional to the number of revocations. We formalize the notion of *key-updating schemes for lazy revocation* and give a rigorous security definition. In our model, a *center* (e.g., the group owner) initially generates some state information, which takes the role of the master secret key. The center state is updated at every revocation. We call the period of time between two revocations a *time interval*. Upon a user request, the center uses its current local state to derive a *user key* and gives that to the user. From the user key of some time interval, a user must be able to extract the key for any previous time interval efficiently. Security for key-updating schemes requires that any polynomial-time adversary with access to the user key for a particular time interval does not obtain any information about the keys for future time intervals. The keys generated by our key-updating schemes can be used with a symmetric encryption algorithm to encrypt files for confidentiality or with a message-authentication code to authenticate files for integrity protection.

We describe two generic composition methods that combine two secure key updating schemes into a new scheme in which the number of time intervals is either the sum or the product of the number of time intervals of the initial schemes. Additionally, we investigate three constructions of key-updating schemes. The first scheme uses a chain of pseudorandom generator applications and is related to existing methods using one-way hash chains. It has constant update cost for the center, but the complexity of the user-key derivation is linear in the total number of time intervals. The second scheme can be based on arbitrary trapdoor permutations and generalizes the key rotation construction of the Plutus file system [21]. It has constant update and user-key derivation times, but the update algorithm uses a relatively expensive public-key operation. These two constructions require that the total number T of time intervals is polynomial in the security parameter. Our third scheme uses a novel construction. It relies on a tree to derive the keys at the leaves from the master key at the root. The tree can be seen as resulting from the iterative application of the additive composition method and supports a practically unbounded number of time intervals. The binary-tree construction balances the tradeoff between the center-state update and user-key derivation algorithms (both of them have logarithmic complexity in T), at the expense of increasing the sizes of the user key and center state by a logarithmic factor in T.

The rest of the paper is organized as follows. In Section 2 we give the definition of security for key-updating schemes. In Section 3, we introduce the additive and multiplicative composition methods for secure key-updating schemes. The three constructions and proofs for their security are presented in Section 4. A systematic analysis of the computational and communication complexities of the three constructions and a comparison with related work are given in Sections 5 and 6, respectively.

2 Definitions

2.1 Key-Updating Schemes

In our model, we divide time into intervals, not necessarily of fixed length, and each time interval is associated with a new key that can be used in a symmetric-key cryptographic algorithm. In a key-updating scheme, the center generates initial state information that is updated at each time interval, and from which the center can derive a user key. The user key for interval t permits a user to derive the keys of previous time intervals (k_i for $i \leq t$), but it should not give any information about keys of future time intervals (k_i for $i > t$).

We formalize key-updating schemes using the approach of modern cryptography and denote the security parameter by κ. For simplicity, we assume that all the keys are bit strings of length κ. The number of time intervals and the security parameter are given as input to the initialization algorithm.

Definition 1 (Key-Updating Schemes). A key-updating scheme consists of four deterministic polynomial time algorithms $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ with the following properties:

- The initialization algorithm, Init, takes as input the *security parameter* 1^κ, the *number of time intervals* T and a *random seed* $s \in \{0,1\}^{l(\kappa)}$ for a polynomial $l(\kappa)$, and outputs a bit string S_0, called the initial center *state*.

- The key update algorithm, Update, takes as input the current *time interval* t, the current center *state* S_t, and outputs the center *state* S_{t+1} for the next time interval.

- The user key derivation algorithm, Derive, is given as input a *time interval* t and the center *state* S_t, and outputs the *user key* M_t. The user key can be used to derive all keys k_i for $1 \leq i \leq t$.

- The key extraction algorithm, $\mathsf{Extract}$, is executed by the user and takes as input a *time interval* t, the *user key* M_t for interval t as received from the center, and a *target time interval* i with $1 \leq i \leq t$. The algorithm outputs the *key* k_i for interval i.

2.2 Security of Key-Updating Schemes

The definition of security for key-updating schemes requires that a polynomial-time adversary with access to the user key for a time interval t is not able to derive any information about the keys for the next time interval. Formally, consider a probabilistic polynomial-time adversary $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ that participates in the following experiment:

Initialization: The initial center state is generated with the Init algorithm.

Key updating: The adversary adaptively picks a time interval t such that $0 \leq t \leq T-1$ as follows. Starting with $t = 0, 1, \ldots$, algorithm $\mathcal{A}_\mathcal{U}$ is given the user keys M_t for all consecutive time intervals until $\mathcal{A}_\mathcal{U}$ decides to output stop or t becomes equal to $T-1$. We require that $\mathcal{A}_\mathcal{U}$, a probabilistic polynomial-time algorithm, outputs stop at least once before halting. $\mathcal{A}_\mathcal{U}$ also outputs some additional information $z \in \{0,1\}^*$ that is given as input to algorithm $\mathcal{A}_\mathcal{G}$.

Challenge: A challenge for the adversary is generated, which is either the key for time interval $t+1$ generated with the Update, Derive and $\mathsf{Extract}$ algorithms, or a random bit string of length κ.

Guess: $\mathcal{A}_\mathcal{G}$ takes the challenge and z as inputs and outputs a bit b.

The key-updating scheme is secure if the advantage of the adversary of distinguishing between the properly generated key for time interval $t+1$ and the random key is only negligibly larger than $\frac{1}{2}$. More formally, the definition of a secure key-updating scheme is the following:

Definition 2 (Security of Key-Updating Schemes). Let $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ be a key-updating scheme and $\mathcal{A}$ a polynomial-time adversary algorithm that participates in one of the two experiments defined in Figure 1. The advantage of the adversary $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ for the key-updating scheme KU is defined as

$$\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(\mathcal{A}) = \left|\Pr\left[\mathsf{Exp}^{\mathtt{sku\text{-}1}}_{\mathsf{KU},\mathcal{A}}(1^\kappa, T) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathtt{sku\text{-}0}}_{\mathsf{KU},\mathcal{A}}(1^\kappa, T) = 1\right]\right|.$$

$\mathsf{Exp}^{\mathsf{sku}\text{-}0}_{\mathsf{KU},\mathcal{A}}(1^\kappa, T)$
$\quad S_0 \leftarrow \mathsf{Init}(1^\kappa, T)$
$\quad t \leftarrow 0$
$\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \perp, \perp)$
$\quad \mathsf{while}(d \neq \mathsf{stop})\ \mathsf{and}\ (t < T - 1)$
$\quad\quad t \leftarrow t + 1$
$\quad\quad S_t \leftarrow \mathsf{Update}(t - 1, S_{t-1})$
$\quad\quad M_t \leftarrow \mathsf{Derive}(t, S_t)$
$\quad\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, M_t, z)$
$\quad S_{t+1} \leftarrow \mathsf{Update}(t, S_t)$
$\quad M_{t+1} \leftarrow \mathsf{Derive}(t + 1, S_{t+1})$
$\quad k_{t+1} \leftarrow \mathsf{Extract}(t + 1, M_{t+1})$
$\quad b \leftarrow \mathcal{A}_\mathcal{G}(k_{t+1}, z)$
$\quad \mathsf{return}\ b$

$\mathsf{Exp}^{\mathsf{sku}\text{-}1}_{\mathsf{KU},\mathcal{A}}(1^\kappa, T)$
$\quad S_0 \leftarrow \mathsf{Init}(1^\kappa, T)$
$\quad t \leftarrow 0$
$\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, \perp, \perp)$
$\quad \mathsf{while}(d \neq \mathsf{stop})\ \mathsf{and}\ (t < T - 1)$
$\quad\quad t \leftarrow t + 1$
$\quad\quad S_t \leftarrow \mathsf{Update}(t - 1, S_{t-1})$
$\quad\quad M_t \leftarrow \mathsf{Derive}(t, S_t)$
$\quad\quad (d, z) \leftarrow \mathcal{A}_\mathcal{U}(t, M_t, z)$
$\quad k_{t+1} \leftarrow_R \{0, 1\}^\kappa$
$\quad b \leftarrow \mathcal{A}_\mathcal{G}(k_{t+1}, z)$
$\quad \mathsf{return}\ b$

Figure 1: Experiments defining the security of key-updating schemes.

Without loss of generality, we can relate the success probability of adversary $\mathcal{A}$ of distinguishing between the two experiments and its advantage as

$$\Pr[\mathcal{A} \text{ succeeds}] = \frac{1}{2}\Big[\Pr\big[\mathsf{Exp}^{\mathsf{sku}\text{-}0}_{\mathsf{KU},\mathcal{A}} = 0\big] + \Pr\big[\mathsf{Exp}^{\mathsf{sku}\text{-}1}_{\mathsf{KU},\mathcal{A}} = 1\big]\Big] = \frac{1}{2}\Big[1 + \mathsf{Adv}^{\mathsf{sku}}_{\mathsf{KU}}(\mathcal{A})\Big]. \tag{1}$$

The maximum advantage of all probabilistic polynomial-time adversaries is denoted

$$\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{KU}} = \max_{\mathcal{A}}\{\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{KU}}(\mathcal{A})\}.$$

The key-updating scheme KU is *secure* if there exists a negligible function ϵ such that $\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{KU}} = \epsilon(\kappa)$.

Remark. The security notion we have defined is equivalent to a seemingly stronger security definition, in which the adversary can choose the challenge time interval t^* with the restriction that t^* is greater than the time interval at which the adversary outputs stop and that t^* is polynomial in the security parameter. This second security definition guarantees, intuitively, that the adversary is not gaining any information about the keys of any future time intervals after it outputs stop.

3 Composition of Key-Updating Schemes

Let $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ and $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ be two secure key-updating schemes using the same security parameter κ with T_1 and T_2 time intervals, respectively. In this section, we show how to combine the two schemes into a secure key-updating scheme $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$, which is either the additive or multiplicative composition of the two schemes with $T = T_1 + T_2$ and $T = T_1 \cdot T_2$ time intervals, respectively. Similar generic composition methods have been given previously for forward-secure signature schemes [24].

For simplicity, we assume the length of the random seed in the Init algorithm of the scheme KU to be κ for both composition methods. Let $G : \{0,1\}^\kappa \to \{0,1\}^{l_1(\kappa)+l_2(\kappa)}$ be a pseudorandom generator; it can be used to expand a random seed of length κ into two random bit strings of length $l_1(\kappa)$ and $l_2(\kappa)$, respectively, as needed for Init_1 and Init_2. We write $G(s) = G_1(s) \| G_2(s)$ with $|G_1(s)| = l_1(\kappa)$ and $|G_2(s)| = l_2(\kappa)$ for $s \in \{0,1\}^\kappa$.

$\mathsf{Init}(1^\kappa, T, s)$ $\quad S_0^1 \leftarrow \mathsf{Init}_1(1^\kappa, T_1, G_1(s))$ $\quad S_0^2 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(s))$ $\quad$ return (S_0^1, S_0^2)	$\mathsf{Derive}(t, (S_t^1, S_t^2))$ $\quad$ if $t < T_1$ $\qquad M_t^1 \leftarrow \mathsf{Derive}_1(t, S_t^1)$ $\qquad M_t^2 \leftarrow \perp$ $\quad$ else $\qquad M_t^1 \leftarrow \mathsf{Derive}_1(T_1, S_t^1)$ $\qquad M_t^2 \leftarrow \mathsf{Derive}_2(t - T_1, S_t^2)$ $\quad$ return(M_t^1, M_t^2)
$\mathsf{Update}(t, (S_t^1, S_t^2))$ $\quad$ if $t < T_1$ $\qquad S_{t+1}^1 \leftarrow \mathsf{Update}_1(t, S_t^1)$ $\qquad S_{t+1}^2 \leftarrow S_t^2$ $\quad$ else $\qquad S_{t+1}^1 \leftarrow S_t^1$ $\qquad S_{t+1}^2 \leftarrow \mathsf{Update}_2(t - T_1, S_t^2)$ $\quad$ return (S_{t+1}^1, S_{t+1}^2)	$\mathsf{Extract}(t, (M_t^1, M_t^2), i)$ $\quad$ if $i > T_1$ $\qquad k_i \leftarrow \mathsf{Extract}_2(t - T_1, M_t^2, i - T_1)$ $\quad$ else $\qquad$ if $t < T_1$ $\qquad\quad k_i \leftarrow \mathsf{Extract}_1(t, M_t^1, i)$ $\qquad$ else $\qquad\quad k_i \leftarrow \mathsf{Extract}_1(T_1, M_t^1, i)$ $\quad$ return k_i

Figure 2: The additive composition of KU_1 and KU_2.

3.1 Additive Composition

The additive composition of two key-updating schemes uses the keys generated by the first scheme for the first T_1 time intervals and the keys generated by the second scheme for the subsequent T_2 time intervals. The user key for the first T_1 intervals in KU is the same as that of scheme KU_1 for the same interval. For an interval t greater than T_1, the user key includes both the user key for interval $t - T_1$ of scheme KU_2, and the user key for interval T_1 of scheme KU_1. The details of the additive composition method are described in Figure 2. The security of the composition operation is analyzed in the following theorem.

Theorem 1. *Suppose that* $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ *and* $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ *are two secure key-updating schemes with* T_1 *and* T_2 *time intervals, respectively, and that* G *is a pseudorandom generator as above. Then* $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ *described in Figure 2 denoted as* $\mathsf{KU}_1 \oplus \mathsf{KU}_2$ *is a secure key-updating scheme with* $T_1 + T_2$ *time intervals.*

Proof. Let $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ be a polynomial-time adversary for KU. We build two adversary algorithms $\mathcal{A}^1 = (\mathcal{A}_\mathcal{U}^1, \mathcal{A}_\mathcal{G}^1)$ and $\mathcal{A}^2 = (\mathcal{A}_\mathcal{U}^2, \mathcal{A}_\mathcal{G}^2)$ for KU_1 and KU_2, respectively.

Construction of $\mathcal{A}^1$. $\mathcal{A}^1$ simulates the environment for $\mathcal{A}$, by giving to $\mathcal{A}_\mathcal{U}$ at each iteration t the user key M_t^1 that $\mathcal{A}_\mathcal{U}^1$ receives from the center. If $\mathcal{A}$ aborts or $\mathcal{A}_\mathcal{U}$ does not output stop until time interval $T_1 - 1$, then $\mathcal{A}^1$ outputs $\perp$ and aborts. Otherwise, $\mathcal{A}_\mathcal{U}^1$ outputs stop at the same time interval as $\mathcal{A}_\mathcal{U}$. In the challenge phase, $\mathcal{A}_\mathcal{G}^1$ receives as input a challenge key k_{t+1} and gives that to $\mathcal{A}_\mathcal{G}$. $\mathcal{A}_\mathcal{G}^1$ outputs the same bit as $\mathcal{A}_\mathcal{G}$. The success probability of $\mathcal{A}^1$ for $b \in \{0, 1\}$ is

$$\Pr\left[\mathsf{Exp}_{\mathsf{KU}_1,\mathcal{A}^1}^{\mathtt{sku}\text{-}b} = b\right] = \Pr\left[\mathsf{Exp}_{\mathsf{KU},\mathcal{A}}^{\mathtt{sku}\text{-}b} = b | E_1 \cap E_2\right], \tag{2}$$

where E_1 is the event that $\mathcal{A}_\mathcal{U}$ outputs stop at a time interval strictly less than T_1 and E_2 the event that $\mathcal{A}$ does not distinguish the simulation done by $\mathcal{A}^1$ from the protocol execution. The only difference between the simulation and the protocol execution is that the initial state for KU_1 is a random seed in the simulation and it is generated using a pseudorandom generator G in the protocol. If $\mathcal{A}$ distinguishes the simulation from the protocol, then a distinguisher algorithm D for the pseudorandom generator with advantage $\mathsf{Adv}_G^{\mathtt{prg}}(D)$ can be constructed. By the definition of E_2, we have $\Pr[\bar{E_2}] = \mathsf{Adv}_G^{\mathtt{prg}}(D)$.

Construction of $\mathcal{A}^2$. $\mathcal{A}^2$ simulates the environment for $\mathcal{A}$: it first picks a random seed s of length κ and generates from $G_1(s)$ an instance of the scheme KU_1. For the first T_1 iterations of $\mathcal{A}_\mathcal{U}$, $\mathcal{A}^2$ gives to $\mathcal{A}_\mathcal{U}$ the user keys generated from s. If $\mathcal{A}$ aborts or $\mathcal{A}_\mathcal{U}$ stops at a time interval less than T_1, then $\mathcal{A}^2$ aborts the simulation. For the next T_2 time interval, $\mathcal{A}^2$ feeds $\mathcal{A}_\mathcal{U}$ the user keys received from the center. If $\mathcal{A}_\mathcal{U}$ outputs stop at a time interval $t \geq T_1$, then $\mathcal{A}^2_\mathcal{U}$ outputs stop at time interval $t - T_1$. In the challenge phase, $\mathcal{A}^2_\mathcal{G}$ receives a challenge k_{t-T_1+1}, gives this challenge to $\mathcal{A}_\mathcal{G}$ and outputs what $\mathcal{A}_\mathcal{G}$ outputs. The success probability of $\mathcal{A}^2$ for $b \in \{0,1\}$ is

$$\Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\right] = \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b | \bar{E}_1 \cap E_2\right]. \quad (3)$$

We can relate the success probabilities of $\mathcal{A}$, $\mathcal{A}^1$, and $\mathcal{A}^2$ for $b \in \{0,1\}$ as follows:

$$\begin{aligned}
\Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b\right] &= \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b \cap E_2\right] + \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b \cap \bar{E}_2\right] \\
&= \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b \cap E_2 \cap E_1\right] + \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b \cap E_2 \cap \bar{E}_1\right] + \\
&\quad \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b \cap \bar{E}_2\right] \\
&\leq \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b | E_1 \cap E_2\right] \Pr\left[E_1 \cap E_2\right] + \\
&\quad \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU},\mathcal{A}} = b | E_2 \cap \bar{E}_1\right] \Pr\left[E_2 \cap \bar{E}_1\right] + \Pr\left[\bar{E}_2\right] \\
&= \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU}_1,\mathcal{A}^1} = b\right] \Pr\left[E_1 \cap E_2\right] + \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\right] \Pr\left[E_2 \cap \bar{E}_1\right] + \\
&\quad \Pr\left[\bar{E}_2\right] \quad (4) \\
&\leq \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU}_1,\mathcal{A}^1} = b\right] + \Pr\left[\mathsf{Exp}^{\texttt{sku-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\right] + \Pr\left[\bar{E}_2\right],
\end{aligned}$$

where (4) follows from (2) and (3). Finally, we can infer from (1)

$$\mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}}(\mathcal{A}) \leq \mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}_1}(\mathcal{A}^1) + \mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}_2}(\mathcal{A}^2) + \mathsf{Adv}^{\texttt{prg}}_{G}(D).$$

Since $\mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}_1}(\mathcal{A}^1)$, $\mathsf{Adv}^{\texttt{sku}}_{\mathsf{KU}_2}(\mathcal{A}^2)$ and $\mathsf{Adv}^{\texttt{prg}}_{G}(D)$ are negligible from the assumptions of the theorem, the statement of the theorem follows. □

Extended Additive Composition. It is immediate to extend the additive composition to construct a new scheme with T_1+T_2+1 time intervals. The idea is to use the first scheme for the keys of the first T_1 intervals, the second scheme for the keys of the next T_2 intervals, and the seed s as the key for the (T_1+T_2+1)-th interval. By revealing the seed s as the user key at interval T_1+T_2+1, all previous keys of KU_1 and KU_2 can be derived. This idea will be useful in our later construction of a binary tree key-updating scheme. We call this composition method *extended additive composition*.

3.2 Multiplicative Composition

The idea behind the multiplicative composition operation is to use every key of the first scheme to seed an instance of the second scheme. Thus, for each one of the T_1 time intervals of the first scheme, we generate an instance of the second scheme with T_2 time intervals.

In the sequel, we denote a time interval t for $1 \leq t \leq T_1 \cdot T_2$ of scheme KU as a pair $t = <i,j>$, where i and j are such that $t = (i-1)T_2 + j$ for $1 \leq i \leq T_1$ and $1 \leq j \leq T_2$. The user key for a time interval $t = <i,j>$ includes both the user key for time interval $i-1$ of scheme KU_1 and the user key for time interval j of scheme KU_2. A user receiving $M_{<i,j>}$ can extract the key for any time interval $<m,n> \leq <i,j>$ by first extracting the key K for time interval m of KU_1 (this step needs to be performed only if $m < i$), then using K to derive the initial state of the m-th instance of the scheme KU_2, and finally, deriving the key $k_{<m,n>}$. The details of the multiplicative composition method are shown in Figure 3. The security of the multiplicative composition method is analyzed in the following theorem.

$\mathsf{Init}(1^\kappa, T, s)$
    $S_0 \leftarrow \mathsf{Init}_1(1^\kappa, T_1, G_1(s))$
    return $(\perp, S_0, S_0)$
	$\mathsf{Derive}(<i, j>, (S^1_{i-1}, S^1_i, S^2_j))$
	    if $i > 1$
	        $M^1_{i-1} \leftarrow \mathsf{Derive}_1(i-1, S^1_{i-1})$
	    else
	        $M^1_{i-1} \leftarrow \perp$
	    $M^2_j \leftarrow \mathsf{Derive}_2(j, S^2_j)$
	    return (M^1_{i-1}, M^2_j)

$\mathsf{Update}(<i, j>, (S^1_{i-1}, S^1_i, S^2_j))$
    if $j = T_2$
        $S^1_{i+1} \leftarrow \mathsf{Update}_1(i, S^1_i)$
        $k^1_{i+1} \leftarrow \mathsf{Extract}_1(i+1,$
                $\mathsf{Derive}_1(i+1, S^1_{i+1}), i+1)$
        $S^2_0 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k^1_{i+1}))$
        $S^2_1 \leftarrow \mathsf{Update}_2(0, S^2_0)$
        return $(S^1_i, S^1_{i+1}, S^2_1)$
    else
        $S^2_{j+1} \leftarrow \mathsf{Update}_2(j, S^2_j)$
        return $(S^1_{i-1}, S^1_i, S^2_{j+1})$
	$\mathsf{Extract}(<i, j>, (M^1_{i-1}, M^2_j), <m, n>)$
	    if $i = m$
	        $k_{<m,n>} \leftarrow \mathsf{Extract}_2(j, M^2_j, m)$
	    else
	        $K \leftarrow \mathsf{Extract}_1(i-1, M^1_{i-1}, m)$
	        $S^2_0 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(K))$
	        $k_{<m,n>} \leftarrow \mathsf{Extract}_2(T_2, S^2_0, n)$
	    return $k_{<m,n>}$

Figure 3: The multiplicative composition of KU_1 and KU_2.

Theorem 2. *Suppose that* $\mathsf{KU}_1 = (\mathsf{Init}_1, \mathsf{Update}_1, \mathsf{Derive}_1, \mathsf{Extract}_1)$ *and* $\mathsf{KU}_2 = (\mathsf{Init}_2, \mathsf{Update}_2, \mathsf{Derive}_2, \mathsf{Extract}_2)$ *are two secure key-updating schemes with* T_1 *and* T_2 *time intervals, respectively, and that* G *is a pseudorandom generator as above. Then* $\mathsf{KU} = (\mathsf{Init}, \mathsf{Update}, \mathsf{Derive}, \mathsf{Extract})$ *described in Figure 3 denoted as* $\mathsf{KU}_1 \otimes \mathsf{KU}_2$ *is a secure key-updating scheme with* $T_1 \cdot T_2$ *time intervals.*

Proof. Let $\mathcal{A} = (\mathcal{A}_\mathcal{U}, \mathcal{A}_\mathcal{G})$ be a polynomial-time adversary for KU. Similarly to the proof of Theorem 1, we build two adversary algorithms $\mathcal{A}^1 = (\mathcal{A}^1_\mathcal{U}, \mathcal{A}^1_\mathcal{G})$ and $\mathcal{A}^2 = (\mathcal{A}^2_\mathcal{U}, \mathcal{A}^2_\mathcal{G})$ for KU_1 and KU_2, respectively.

Construction of $\mathcal{A}^1$. $\mathcal{A}^1_\mathcal{U}$ gets from the center the user keys M^1_i of scheme KU_1 for all time intervals i until it outputs stop. $\mathcal{A}^1$ simulates the environment for $\mathcal{A}$ by sending the following user keys:

1. At interval $<i, 1>$, for $1 \leq i \leq T_1$, $\mathcal{A}^1$ runs $k_i \leftarrow \mathsf{Extract}_1(i, M^1_i, i)$; $S^2_0 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_i))$; $S^2_1 \leftarrow \mathsf{Update}_2(0, S^2_0)$; $M^2_1 \leftarrow \mathsf{Derive}_2(1, S^2_1)$ and gives $\mathcal{A}_\mathcal{U}$ the user key $M_{<i,1>} = (M^1_{i-1}, M^2_1)$.
2. At time interval $<i, j>$, for $1 \leq i \leq T_1$ and $1 < j \leq T_2$, $\mathcal{A}^1_\mathcal{U}$ computes $S^2_j \leftarrow \mathsf{Update}_2(j-1, S^2_{j-1})$ and $M^2_j \leftarrow \mathsf{Derive}_2(j, S^2_j)$ and gives to $\mathcal{A}_\mathcal{U}$ the user key $M_{<i,j>} = (M^1_{i-1}, M^2_j)$.

If $\mathcal{A}$ aborts or $\mathcal{A}_\mathcal{U}$ outputs stop at a time interval $<i, j>$ with $j \neq T_2$, then $\mathcal{A}^1_\mathcal{U}$ aborts the simulation and outputs $\perp$. Otherwise, $\mathcal{A}^1_\mathcal{U}$ outputs stop at time interval i. In the challenge interval, $\mathcal{A}^1_\mathcal{G}$ is given a challenge key k_{i+1} and it executes $S^2_0 \leftarrow \mathsf{Init}_2(1^\kappa, T_2, G_2(k_{i+1}))$; $S^2_1 \leftarrow \mathsf{Update}_2(0, S^2_0)$; $M \leftarrow \mathsf{Derive}_2(1, S^2_1)$; $k^2_1 \leftarrow \mathsf{Extract}_2(1, M, 1)$. It then gives the challenge k^2_1 to $\mathcal{A}_\mathcal{G}$. $\mathcal{A}^1_\mathcal{G}$ outputs the same bit as $\mathcal{A}_\mathcal{G}$. The success probability of $\mathcal{A}^1$ for $b \in \{0, 1\}$ is

$$\Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1, \mathcal{A}^1} = b\right] = \Pr\left[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}, \mathcal{A}} = b \mid E_1 \cap E_2\right], \tag{5}$$

where E_1 is the event that $\mathcal{A}_\mathcal{U}$ outputs stop at a time interval (i, j) with $j = T_2$ and E_2 the event that $\mathcal{A}$ does not distinguish the simulation done by $\mathcal{A}^1$ from the protocol execution. If $\mathcal{A}$ distinguishes the simulation from the protocol, then a distinguisher algorithm D for the pseudorandom generator with advantage $\mathsf{Adv}^{\mathtt{prg}}_G(D)$ can be constructed. By the definition of E_2, we have $\Pr[\bar{E}_2] = \mathsf{Adv}^{\mathtt{prg}}_G(D)$.

Construction of $\mathcal{A}^2$. Assuming that $\mathcal{A}_{\mathcal{U}}$ runs at most q times (and q is polynomial in κ), $\mathcal{A}^2$ makes a guess for the time interval i^* in which $\mathcal{A}_{\mathcal{U}}$ outputs stop. $\mathcal{A}^2$ picks i^* uniformly at random from the set $\{1, \ldots, q\}$. $\mathcal{A}^2$ generates an instance of the scheme KU_1 with i^* time intervals. For any interval $<i, j>$ with $i < i^*$, $\mathcal{A}^2$ generates the user keys using the keys from this instance of KU_1. For time intervals $<i^*, j>$ with $1 \leq j \leq T_2$, $\mathcal{A}^2$ outputs user key $(M^1_{i^*-1}, M^2_j)$, where $M^1_{i^*-1}$ is the user key for time interval $i^* - 1$ of KU_1 that it generated itself and M^2_j is the user key for time interval j of KU_2 that it received from the center.

If $\mathcal{A}$ aborts or $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $<i, j>$ with $i \neq i^*$ or with $i = i^*$ and $j = T_2$, then $\mathcal{A}^2$ aborts the simulation and outputs $\perp$. Otherwise, if $\mathcal{A}_{\mathcal{U}}$ outputs stop at a time interval $<i^*, j>$, then $\mathcal{A}^2_{\mathcal{U}}$ outputs stop at time interval j. In the challenge phase, $\mathcal{A}^2$ receives a challenge key k_{j+1} and gives that to $\mathcal{A}_{\mathcal{G}}$. $\mathcal{A}^2_{\mathcal{G}}$ outputs the same bit as $\mathcal{A}_{\mathcal{G}}$. The success probability of $\mathcal{A}^2$ for $b \in \{0, 1\}$ is

$$\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\big] = \frac{1}{q}\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}} = b|\bar{E}_1 \cap E_2\big]. \tag{6}$$

As in the proof of Theorem 1, we can infer

$$\begin{aligned}
\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}} = b\big] &\leq \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}} = b|E_1 \cap E_2\big]\Pr\big[E_1 \cap E_2\big] + \\
&\quad \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU},\mathcal{A}} = b|\bar{E}_1 \cap E_2\big]\Pr\big[\bar{E}_1 \cap E_2\big] + \Pr[\bar{E}_2] \\
&= \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1} = b\big]\Pr\big[E_1 \cap E_2\big] + q\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\big]\Pr\big[\bar{E}_1 \cap E_2\big] \\
&\quad + \Pr[\bar{E}_2] \\
&\leq \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_1,\mathcal{A}^1} = b\big] + q\Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{KU}_2,\mathcal{A}^2} = b\big] + \Pr[\bar{E}_2],
\end{aligned} \tag{7}$$

where (7) follows from (5) and (6). Finally we can infer from (1) that

$$\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1}(\mathcal{A}^1) + q\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2}(\mathcal{A}^2) + \mathsf{Adv}^{\mathtt{prg}}_{G}(D).$$

Since $\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_1}(\mathcal{A}^1)$, $\mathsf{Adv}^{\mathtt{sku}}_{\mathsf{KU}_2}(\mathcal{A}^2)$ and $\mathsf{Adv}^{\mathtt{prg}}_{G}(D)$ are negligible from the assumptions of the theorem, the statement of the theorem follows. □

4 Constructions

In this section, we describe three constructions of key-updating schemes with different complexity and communication tradeoffs. The first two constructions are based on previously proposed methods, whose security has never been formally analyzed. We give cryptographic proofs that demonstrate the security of the existing constructions after some subtle modifications. Additionally, we propose a third construction that is more efficient than the known schemes. It uses a binary tree to derive the user keys and is also provably secure in our model.

4.1 Chaining Construction (CKU)

In this construction the user keys and keys are generated iteratively from a random seed using a pseudorandom generator $G : \{0, 1\}^\kappa \rightarrow \{0, 1\}^{2\kappa}$. We write $G(s) = G_1(s)\|G_2(s)$ with $|G_1(s)| = |G_2(s)| = \kappa$ for $s \in \{0, 1\}^\kappa$. The algorithms of the chaining construction, called CKU, are the following:

- $\mathsf{Init}(1^\kappa, T, s)$ generates a random seed s_0 of length κ from s and outputs $S_0 = s_0$.

- $\mathsf{Update}(t, S_t)$ copies the state S_t into S_{t+1}.

- $\mathsf{Derive}(t, S_t)$ and $\mathsf{Extract}(t, M_t, i)$ are given in Figure 4.

Derive(t, S_t)	Extract(t, M_t, i)
$B_{T+1} \leftarrow S_t$ for $i = T$ downto t $(B_i, k_i) \leftarrow G(B_{i+1})$ return (B_t, k_t)	$(B_t, k_t) \leftarrow M_t$ for $j = t-1$ downto i $(B_j, k_j) \leftarrow G(B_{j+1})$ return k_i

Figure 4: The Derive(t, S_t) and Extract(t, M_t, i) algorithms of the chaining construction.

This construction has constant center-state size and linear cost for the user-key derivation algorithm. An alternative construction with linear center-state size and constant user-key derivation is to precompute all the keys k_i and user keys M_i, for $1 \le i \le T$ in the Init algorithm and store all of them in the initial center state S_0.

Theorem 3. *Given a pseudorandom generator G,* CKU *is a secure key-updating scheme.*

Proof. Let $\mathcal{A} = (\mathcal{A}_{\mathcal{U}}, \mathcal{A}_{\mathcal{G}})$ be a polynomial-time adversary successful in breaking the security of the key-updating scheme. We construct an algorithm D that distinguishes the output of the pseudorandom generator from a random string of length 2κ with sufficiently large probability.

Algorithm D has to simulate the environment for $\mathcal{A}$. D picks B_{T+1} uniformly at random from $\{0,1\}^\kappa$ and computes the user keys for previous time intervals as $(B_i, k_i) = G(B_{i+1})$, for $i = T, \ldots, 1$. D gives to $\mathcal{A}_{\mathcal{U}}$ user key $M_i = (B_i, k_i)$ at iteration i.

Algorithm D is given a challenge string $r = r_0 \| r_1$ of length 2κ, which in experiment 0 is the output of the pseudorandom generator on input a random seed of length κ, and in experiment 1 is a random string of length 2κ. Formally, the prg experiments are defined in Figure 5.

$\mathsf{Exp}^{\mathtt{prg}\text{-}0}_{G,D}$	$\mathsf{Exp}^{\mathtt{prg}\text{-}1}_{G,D}$
$s \leftarrow_R \{0,1\}^\kappa$ $r_0 \| r_1 \leftarrow G(s)$ $b \leftarrow D(r_0 \| r_1)$ return b	$r_0 \| r_1 \leftarrow_R \{0,1\}^{2\kappa}$ $b \leftarrow D(r_0 \| r_1)$ return b

Figure 5: Experiments defining the security of pseudorandom generator G.

If $\mathcal{A}_{\mathcal{U}}$ outputs stop at time interval t, D gives to $\mathcal{A}_{\mathcal{G}}$ the challenge key $k_{t+1} = r_1$ and D outputs what $\mathcal{A}_{\mathcal{G}}$ outputs. Denote by $p_b = \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}b}_{\mathsf{CKU},\mathcal{A}} = b\big]$. It is immediate that

$$\Pr\big[\mathsf{Exp}^{\mathtt{prg}\text{-}1}_{G,D} = 1\big] = \Pr\big[\mathsf{Exp}^{\mathtt{sku}\text{-}1}_{\mathsf{CKU},\mathcal{A}} = 1\big] = p_1, \tag{8}$$

and

$$\Pr\big[\mathsf{Exp}^{\mathtt{prg}\text{-}0}_{G,D} = 0\big] = p_0', \tag{9}$$

where p_0' is the probability that $\mathcal{A}$, given the user keys as in experiment $\mathsf{Exp}^{\mathtt{sku}\text{-}0}$, but challenge key $k_{t+1} = G_2(s)$ for a random seed $s \in \{0,1\}^\kappa$, outputs 0. The challenge key given to $\mathcal{A}$ in experiment $\mathsf{Exp}^{\mathtt{sku}\text{-}0}$ is $G_2(G_1^{T-t-1}(s))$, where $G_1^i(s) = G_1(\ldots G_1(s) \ldots)$ for i applications of G_1. We can bound the absolute difference between p_0 and p_0' as

$$\begin{aligned} |p_0' - p_0| &\le \Pr\big[\mathcal{A} \text{ distinguishes between } G_2(s) \text{ and } G_2(G_1^{T-t-1}(s))\big] \\ &\le (T-t)\Pr\big[\mathcal{A} \text{ distinguishes between } s \leftarrow_R \{0,1\}^\kappa \text{ and } G_1(s)\big] \\ &\le (T-t)\mathsf{Adv}^{\mathtt{prg}}_G. \end{aligned} \tag{10}$$

Using (8), (9) and (10), we can relate the success probabilities of $\mathcal{A}$ and D by

$$\begin{aligned}
\Pr[D \text{ succeeds}] &= \frac{1}{2}\left(\Pr[\mathsf{Exp}^{\mathsf{prg}\text{-}0}_{G,D} = 0] + \Pr[\mathsf{Exp}^{\mathsf{prg}\text{-}1}_{G,D} = 1]\right) \\
&= \frac{1}{2}(p_0' + p_1) \\
&= \frac{1}{2}(p_0 + p_1 + p_0' - p_0) \\
&\geq \Pr[\mathcal{A} \text{ succeeds}] - \frac{1}{2}(T-t)\mathsf{Adv}^{\mathsf{prg}}_G.
\end{aligned}$$

It follows that

$$\Pr[\mathcal{A} \text{ succeeds}] \leq \Pr[D \text{ succeeds}] + \frac{1}{2}(T-t)\mathsf{Adv}^{\mathsf{prg}}_G,$$

and

$$\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{CKU}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{prg}}_G(D) + (T-t)\mathsf{Adv}^{\mathsf{prg}}_G \leq T\mathsf{Adv}^{\mathsf{prg}}_G.$$

The statement of the theorem follows from the fact that $\mathsf{Adv}^{\mathsf{prg}}_G$ is negligible. □

4.2 Trapdoor Permutation Construction (TDKU)

In this construction, the center picks an initial random state that is updated at each time interval by applying the inverse of a trapdoor permutation. The trapdoor is known only to the center, but a user, given the state at a certain moment, can apply the permutation iteratively to generate all previous states. The key for a time interval is generated by applying a hash function, modeled as a random oracle, to the current state. This idea underlies the key rotation mechanism of the Plutus file system [21], with the difference that Plutus uses the output of an RSA trapdoor permutation directly for the encryption key. We could not prove the security of this scheme in our model for key-updating schemes, even when the trapdoor permutation is not arbitrary, but instantiated with the RSA permutation.

This construction has the advantage that knowledge of the total number of time intervals is not needed in advance; on the other hand, its security can only be proved in the random oracle model. Let a family of trapdoor permutations be given such that the domain size of the permutations with security parameter κ is $l(\kappa)$, for some polynomial l. Let $h : \{0,1\}^{l(\kappa)} \rightarrow \{0,1\}^{\kappa}$ be a hash function modeled as a random oracle. The detailed construction of the trapdoor permutation scheme, called TDKU, is presented below:

- $\mathsf{Init}(1^\kappa, T, s)$ generates a random $s_0 \leftarrow_R \{0,1\}^{l(\kappa)}$ and a trapdoor permutation $f : \{0,1\}^{l(\kappa)} \rightarrow \{0,1\}^{l(\kappa)}$ with trapdoor τ from seed s using a pseudorandom generator. Then it outputs $S_0 = (s_0, f, \tau)$.
- $\mathsf{Update}(t, S_t)$ with $S_t = (s_t, f, \tau)$ computes $s_{t+1} = f^{-1}(s_t)$ and outputs $S_{t+1} = (s_{t+1}, f, \tau)$.
- $\mathsf{Derive}(t, S_t)$ outputs $M_t \leftarrow (s_t, f)$.
- $\mathsf{Extract}(t, M_t, i)$ applies the permutation iteratively $t-i$ times to generate state $s_i = f^{t-i}(M_t)$ and then outputs $h(s_i)$.

Theorem 4. *Given a family of trapdoor permutations and a hash function h,* TDKU *is a secure key-updating scheme in the random oracle model.*

Proof. Let $\mathcal{A} = (\mathcal{A}_{\mathcal{U}}, \mathcal{A}_{\mathcal{G}})$ be a polynomial-time adversary successful in breaking the security of the key-updating scheme. Assuming that $\mathcal{A}_{\mathcal{U}}$ runs at most q times, we construct an algorithm $\mathcal{I}$, which given f and $y \leftarrow f(x)$ with $x \leftarrow_R \{0,1\}^{l(\kappa)}$ computes $f^{-1}(y)$ with sufficiently large probability.

Algorithm $\mathcal{I}$ has to simulate the environment for $\mathcal{A}$. $\mathcal{I}$ makes a guess at the time interval t^* in which $\mathcal{A}_\mathcal{U}$ outputs stop. $\mathcal{I}$ picks t^* uniformly at random from the set $\{1, \ldots, q\}$. If $\mathcal{A}_\mathcal{U}$ does not output stop at time interval t^*, then $\mathcal{I}$ aborts the simulation. Otherwise, at time interval t less than t^*, $\mathcal{I}$ gives to $\mathcal{A}_\mathcal{U}$ the user key $M_t = (f^{t^*-t}(y), f)$.

Algorithm Extract is executed by $\mathcal{A}$ as in the description of the scheme, but $\mathcal{I}$ simulates the random oracle for $\mathcal{A}$. If $\mathcal{A}$ queries x to the random oracle for which $f(x) = y$, then $\mathcal{I}$ outputs x. Let E be the event that $\mathcal{A}$ asks query $x = f^{-1}(y)$ to the oracle and $\bar{E}$ the negation of this event. Since the adversary has no advantage in distinguishing the properly generated key k_{t+1} from a randomly generated key if it does not query the random oracle at x, it follows that

$$\Pr[\mathcal{A} \text{ succeeds } | \bar{E}] \leq \frac{1}{2},$$

from which we can infer

$$\Pr[\mathcal{A} \text{ succeeds }] = \Pr[\mathcal{A} \text{ succeeds } | E] \Pr[E] + \Pr[\mathcal{A} \text{ succeeds } | \bar{E}] \Pr[\bar{E}] \leq \Pr[E] + \frac{1}{2}. \quad (11)$$

Equations (1) and (11) imply that $\Pr[E] \geq \frac{1}{2}\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{TDKU}}(\mathcal{A})$. Then the success probability of algorithm $\mathcal{I}$ is at least $\frac{1}{q}\Pr[E] \geq \frac{1}{2q}\mathsf{Adv}^{\mathsf{sku}}_{\mathsf{TDKU}}(\mathcal{A})$. The statement of the theorem follows from the fact that algorithm $\mathcal{I}$ has only a negligible probability of success. □

4.3 Tree Construction (TreeKU)

In the two schemes above, at least one of the algorithms Update, Derive and Extract has worst-case complexity linear in the total number of time intervals. We present a tree construction based on ideas of Canetti, Halevi and Katz [9] with constant complexity for the Derive algorithm and logarithmic worst-case complexity in the number of time intervals for the Update and Extract algorithms. Moreover, the amortized complexity of the Update algorithm is constant. In this construction, the user key size is increased by at most a logarithmic factor in T compared to the user key size of the two constructions described above.

Our tree-based key-updating scheme, called TreeKU, generates keys using a complete binary tree with T nodes, assuming w.l.o.g. that $T = 2^d - 1$ for some $d \in \mathbb{Z}$. Each node in the tree is associated with a time interval between 1 and T, a unique label in $\{0,1\}^*$, a *tree-key* in $\{0,1\}^\kappa$ and an *external key* in $\{0,1\}^\kappa$ such that:

1. Time intervals are assigned to tree nodes using post-order tree traversal, i.e., a node corresponds to interval i if it is the i-th node in the post-order traversal of the tree. We refer to the node associated with interval t as node t.

2. We define a function label that maps node t with $1 \leq t \leq T$ to its label in $\{0,1\}^*$ as follows. The root of the tree is labeled by the empty string ε, and the left and right children of a node with label ℓ are labeled by $\ell\|0$ and by $\ell\|1$, respectively. The parent of a node with label ℓ is denoted by $\mathsf{parent}(\ell)$, thus $\mathsf{parent}(\ell\|0) = \mathsf{parent}(\ell\|1) = \ell$. We denote the length of a label ℓ by $|\ell|$.

3. The tree-key for the root node is chosen at random. The tree-keys for the two children of an internal node in the tree are derived from the tree-key of the parent node using a pseudorandom generator $G : \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$. For an input $s \in \{0,1\}^\kappa$, we write $G(s) = G_1(s)\|G_2(s)$ with $|G_1(s)| = |G_2(s)| = \kappa$. If the tree-key for the internal node with label ℓ is denoted u_ℓ, then the tree-keys for its left and right children are $u_{\ell\|0} = G_0(u_\ell)$ and $u_{\ell\|1} = G_1(u_\ell)$, respectively. This implies that once the tree-key for a node is revealed, then the tree-keys of its children can be computed, but knowing the tree-keys of both children of a node does not reveal any information about the tree-key of the node.

```
Update(t, (P_t, L_t))
    if t = 0
        P_1 ← leftkeys(ε, u_T)                     /* P_1 contains the label/tree-key pairs of all the left-most nodes */
        L_1 ← ∅                                    /* the set of left siblings is empty */
    else
        ℓ_t ← label(t)                             /* compute the label of node t */
        u_t ← searchkey(ℓ_t, P_t)                  /* compute the tree-key of node t */
        if ℓ_t ends in 0                           /* t is the left child of its parent */
            (ℓ_s, u_s) ← rightsib(ℓ_t, P_t)        /* compute the label/tree-key pair of the right sibling of t */
            P_{t+1} ← P_t \ {(ℓ_t, u_t)} ∪ leftkeys(ℓ_s, u_s)   /* update the label/tree-key pairs in P_{t+1} */
            L_{t+1} ← L_t ∪ {(ℓ_t, u_t)}           /* add the label/tree-key pair of t to set of left siblings for t + 1 */
        else                                       /* t is the right child of its parent */
            (ℓ_s, u_s) ← leftsib(ℓ_t, L_t)         /* compute the label/tree-key pair of the left sibling of t */
            P_{t+1} ← P_t \ {(ℓ_t, u_t)}           /* remove label/tree-key pair of t from P_{t+1} */
            L_{t+1} ← L_t \ {(ℓ_s, u_s)}           /* remove label/tree-key pair of left sibling of t from L_{t+1} */
    return (P_{t+1}, L_{t+1})

leftkeys(ℓ, u)
    A ← ∅                                          /* initialize set A with the empty set */
    while |ℓ| ≤ d                                  /* advance to the left until we reach a leaf */
        A ← A ∪ {(ℓ, u)}                           /* add the label and tree-key of the current node in A */
        ℓ ← ℓ||0                                   /* move to left child of the node with label p */
        u ← G_0(u)                                 /* compute the tree-key of the left child */
    return A
```

Figure 6: The $\mathsf{Update}(t, (P_t, L_t))$ algorithm.

4. The external key of a node t is the key k_t output by the scheme to the application for interval t. For a node t with tree-key $u_{\mathsf{label}(t)}$, the external key k_t is obtained by computing $F_{u_{\mathsf{label}(t)}}(1)$, where $F_u(b) = F(u, b)$ and $F : \{0,1\}^\kappa \times \{0,1\} \to \{0,1\}^\kappa$ is a pseudorandom function on bits.

We describe the four algorithms of the binary tree key-updating scheme:

- $\mathsf{Init}(1^\kappa, T, s)$ generates the tree-key for the root node randomly, $u_T \leftarrow_R \{0,1\}^\kappa$, using seed s, and outputs $S_0 = (\{(\varepsilon, u_T)\}, \emptyset)$.
- $\mathsf{Update}(t, S_t)$ updates the state $S_t = (P_t, L_t)$ to the next center state $S_{t+1} = (P_{t+1}, L_{t+1})$. The center state for interval t consists of two sets: P_t that contains pairs of (label, tree-key) for all nodes on the path from the root to node t (including node t), and L_t that contains label/tree-key pairs for all left siblings of the nodes in P_t that are not in P_t.

  We use several functions in the description of the Update algorithm. For a label ℓ and a set A of label/tree-key pairs, we define a function $\mathsf{searchkey}(\ell, A)$ that outputs a tree-key u for which $(\ell, u) \in A$, if the label exists in the set, and $\perp$ otherwise. Given a label ℓ and a set of label/tree-key pairs A, function $\mathsf{rightsib}(\ell, A)$ returns the label and the tree-key of the right sibling of the node with label ℓ, and, similarly, function $\mathsf{leftsib}(\ell, A)$ returns the label and the tree-key of the left sibling of the node with label ℓ (assuming the labels and tree-keys of the siblings are in A). The function leftkeys is given as input a label/tree-key pair of a node and returns all label/tree-key pairs of the left-most nodes in the subtree rooted at the input node, including label and tree-key of the input node.

  The code for the Update and leftkeys algorithms is given in Figure 6. We omit the details of functions searchkey, rightsib and leftsib. The Update algorithm distinguishes three cases:

  1. If $t = 0$, the Update algorithm computes the label/tree-key pairs of all left-most nodes in the

```
Extract(t, M_t, i)
  ℓ_1 ... ℓ_s ← label(i)                          /* the label of i has length s */
  v ← s
  ℓ ← ℓ_1 ... ℓ_v
  while v > 0 and searchkey(ℓ, M_t) = ⊥           /* find a predecessor of i that is in M_t */
    v ← v − 1
    ℓ ← ℓ_1 ... ℓ_v
  for j = v + 1 to s                              /* compute tree-keys of all nodes on path from predecessor to i */
    u_{ℓ_1...ℓ_j} ← G_{ℓ_j}(u_{ℓ_1...ℓ_{j−1}})
  k_{ℓ_1...ℓ_s} ← F_{u_{ℓ_1...ℓ_s}}(1)             /* return external key of node i */
  return k_{ℓ_1...ℓ_s}
```

Figure 7: The $\mathsf{Extract}(t, M_t, i)$ algorithm.

complete tree using function leftkeys and stores them in P_1. The set L_1 is empty in this case, as nodes in P_1 do not have left siblings.

2. If t is the left child of its parent, the successor of node t in post-order traversal is the left-most node in the subtree rooted at the right sibling t' of node t. P_{t+1} contains all label/tree-key pairs in P_t except the tuple for node t, and, in addition, all label/tree-key pairs for the left-most nodes in the subtree rooted at t', which are computed by leftkeys. The set of left siblings L_{t+1} contains all label/tree-key pairs from L_t and, in addition, the label/tree-key pair for node t.

3. If t is the right child of its parent, node $t+1$ is its parent, so P_{t+1} contains all label/tree-key pairs from P_t except the tuple for node t, and L_{t+1} contains all the label/tree-key pairs in L_t except the pair for the left sibling of node t.

- Algorithm $\mathsf{Derive}(t, (P_t, L_t))$ outputs the user tree-key M_t, which is the minimum information needed to generate the set of tree-keys $\{u_i : i \leq t\}$. Since the tree-key of any node reveals the tree-keys for all nodes in the subtree rooted at that node, M_t consists of the label/tree-key pairs for the left siblings (if any) of all nodes on the path from the root to the parent of node t and the label/tree-key pair of node t. This information has already been pre-computed such that one can set $M_t \leftarrow \{(\mathsf{label}(t), u_t)\} \cup L_t$.

- Algorithm $\mathsf{Extract}(t, M_t, i)$ first finds the maximum predecessor of node i in post-order traversal whose label/tree-key pair is included in the user tree-key M_t. Then it computes the tree-keys for all nodes on the path from that predecessor to node i. The external key k_i is derived from the tree-key u_i as $k_i = F_{u_i}(1)$ using the pseudorandom function. The algorithm is in Figure 7.

Analysis of Complexity. The worst-case complexity of the cryptographic operations used in the Update and Extract algorithms is logarithmic in the number of time intervals, and that of Derive is constant. However, it is easy to see that the key for each node is computed exactly once if T updates are executed. This implies that the total cost of all update operations is T pseudorandom-function applications, so the amortized cost per update is constant.

Now we prove the security of the binary tree construction.

Theorem 5. *Given a pseudorandom generator G and a pseudorandom function F, TreeKU is a secure key-updating scheme.*

Proof. Scheme TreeKU with $T = 2^d - 1$ time intervals can be obtained from d extended additive compositions of a trivial key-updating scheme TrivKU with one time interval, defined as follows:

	CKU	TDKU	TreeKU
$\mathsf{Update}(t, S_t)$ time	0	1 PK op.	$\mathcal{O}(\log T)$ PRG op.*
$\mathsf{Derive}(t, S_t)$ time	$T - t$ PRG op.	0	0
$\mathsf{Extract}(t, M_t, i)$ time	$t - i$ PRG op.	$t - i$ PK op.	$\mathcal{O}(\log T)$ PRG op.
Center state size	κ	poly(κ)	$\mathcal{O}(\kappa \log T)$
User key size	κ	κ	$\mathcal{O}(\kappa \log T)$

Figure 8: Worst-case time and space complexities of the constructions. *Note: the amortized complexity of $\mathsf{Update}(t, S_t)$ in the binary tree scheme is constant.

- $\mathsf{Init}(1^\kappa, T, s)$ generates a random user key $M \leftarrow_R \{0,1\}^\kappa$ from the seed s and outputs $S_0 = M$.
- $\mathsf{Update}(t, S_t)$ outputs $S_{t+1} \leftarrow S_t$ only for $t = 0$.
- $\mathsf{Derive}(t, S_t)$ outputs $M_t \leftarrow M$ for $t = 1$.
- $\mathsf{Extract}(t, M_t, i)$ returns $k = F_M(1)$ for $t = i = 1$.

Given that F is a pseudorandom function, it is easy to see that TrivKU is a secure key-updating scheme. Consider an adversary A that has a non-negligible advantage in breaking TrivKU. Since the scheme has one time interval, A is not given any user keys and it has to output stop at time interval 0. We build a distinguisher algorithm D for the pseudorandom function. D is given access to an oracle $G : \{0,1\} \rightarrow \{0,1\}^\kappa$, which is either $F(k, \cdot)$ with $k \leftarrow_R \{0,1\}^\kappa$, or a random function $g \leftarrow_R \{f : \{0,1\} \rightarrow \{0,1\}^\kappa\}$. D gives to A the challenge $k_1 = G(1)$ and outputs the same bit as A. It is immediate that the advantage of D in distinguishing the pseudorandom function from random functions is the same as the advantage of adversary A in breaking TrivKU.

The tree scheme with T time intervals can be constructed as follows: generate 2^{d-1} instances of TrivKU and make them leaves in the tree; build the tree bottom-up by additively composing (using the extended method) two adjacent nodes at the same level in the tree. The security of the binary tree scheme obtained by additive composition as described above follows from Theorem 1. □

5 Performance of the Constructions

In this section we analyze the complexity of the cryptographic operations in the four algorithms and the space complexities of the center state and the user keys for all three proposed constructions. Recall that all schemes generate keys of length κ. In analyzing the time complexity of the algorithms, we specify what kind of operations we measure and distinguish public-key operations (PK op.) from pseudorandom generator applications (PRG op.) because PK operations are typically much more expensive than PRG applications. We omit the time complexity of the Init algorithm, as it involves only the pseudorandom generator for all schemes except for the trapdoor permutation scheme, in which Init also generates the trapdoor permutation. The space complexities are measured in bits. The detailed analysis is in Figure 8.

The chaining scheme CKU has efficient Update and Extract algorithms, but the complexity of the user-key derivation algorithm is linear in the number of time intervals. On the other hand, the trapdoor permutation scheme TDKU has efficient user-key derivation, but the complexity of the Update algorithm is one application of the trapdoor permutation inverse and that of the $\mathsf{Extract}(t, M_t, i)$ algorithm is $t-i$ applications of the trapdoor permutation. The tree-based scheme TreeKU balances the tradeoffs between the complexity of the three algorithms: the cost of Derive algorithm is constant and that of the Update and Extract algorithms is logarithmic in the number of time intervals in the worst-case, at the expense of increasing the center-state and user-key sizes to $\mathcal{O}(\kappa \log T)$. Moreover, the amortized cost of the Update algorithm in the binary tree construction is constant.

Both CKU and TreeKU require the number of time intervals to be known in advance; this is not needed for TDKU. As the chaining and the trapdoor permutation schemes have worst-case complexities linear in T for at least one algorithm, both of them require the number of time intervals to be rather small. In contrast, the binary tree construction can be used for a practically unbounded number of time intervals.

In practical applications, such as key management for cryptographic storage systems, we recommend using a construction similar to the generic forward-secure signature scheme with practically unbounded number of time periods of Malkin, Micciancio, and Miner [24]. The idea is to construct the multiplicative composition of the chaining scheme with binary tree schemes of different sizes. At time interval i of the chaining scheme, the center generates an instance of the binary tree scheme with $2^i - 1$ time intervals. In addition to allowing a practically unbounded number of time intervals, this construction has the property that the complexity of the Update, Derive and Extract algorithms increases with the number of past time intervals.

6 Related Work

Time-Evolving Cryptography. The notion of secure key-updating schemes is closely related to forward- and backward-secure cryptographic primitives. Indeed, a secure key-updating scheme is forward-secure as defined originally by Anderson [4], in the sense that it maintains security in the time intervals following a key exposure. However, this is the opposite of the forward security notion formalized by Bellare and Miner [6] and used in subsequent work. Here we use the term forward security to refer to the latter notion.

Time-evolving cryptography protects a cryptographic primitive against key exposure by dividing the time into intervals and using a different secret key for every time interval. Forward-secure primitives protect past uses of the secret key: if a device holding all keys is compromised, the attacker can not have access to past keys. In the case of forward-secure signatures, the attacker can not generate past signatures on behalf of the user, and in the case of forward-secure encryption, the attacker can not decrypt old ciphertexts. There exist many efficient constructions of forward-secure signatures [6, 2, 19] and several generic constructions [22, 24]. Bellare and Yee [7] analyze forward-secure private-key cryptographic primitives (forward-secure pseudorandom generators, message authentication codes and symmetric encryption) and Canetti, Halevi and Katz [9] construct the first forward-secure public-key encryption scheme.

Forward security has been combined with backward security in models that protect both the past and future time intervals, called key-insulated [12, 13] and intrusion-resilient models [20, 11]. In both models, there is a center that interacts with the user in the key update protocol. The basic key insulation model assumes that the center is trusted and the user is compromised in at most t time intervals and guarantees that the adversary does not gain information about the keys for the intervals the user is not compromised. A variant of this model, called strong key insulation, allows the compromise of the center as well. Intrusion-resilience tolerates arbitrarily many break-ins into both the center and the user, as long as the break-ins do not occur in the same time interval. The relation between forward-secure, key-insulated and intrusion-resilient signatures has been analyzed by Malkin, Obana and Yung [25]. A survey of forward-secure cryptography is given by Itkis [18].

Re-keying, i.e., deriving new secret keys periodically from a master secret key, is a standard method used by many applications. It has been formalized by Abdalla and Bellare [1]. The notion of key-updating schemes that we define is closely related to re-keying schemes, with the difference that in our model, we have the additional requirement of being able to derive past keys efficiently.

Multicast Key Distribution. In key distribution schemes for multicast, a group controller distributes a group encryption key to all users in a multicast group. The group of users is dynamic and each join or leave event requires the change of the encryption key. The goal is to achieve both forward and backward security. In contrast, in our model of key-updating schemes users should be able to derive past encryption keys efficiently.

A common key distribution model for multicast is that of *key graphs*, introduced by Wong et al. [30] and used subsequently in many constructions [28, 27, 17, 16]. In these schemes, each user knows its own secret key and, in addition, a subset of secret keys used to generate the group encryption key and to perform fast update operations. The relation between users and keys is modeled in a directed acyclic graphs, in which the source nodes are the users, intermediary nodes are keys and the unique sink node is the group encryption key. A path from a user node to the group key contains all the keys known to that user. The complexity and communication cost of key update operations is optimal for tree structures [29], and in this case it is logarithmic in the number of users in the multicast group. We also use trees for generating keys, but our approach is different in considering the nodes of the tree to be only keys, and not users. We obtain logarithmic update cost in the number of revocations, not in the number of users in the group.

Key Management in Cryptographic Storage Systems. Early cryptographic file systems [8, 10] did not address key management. Cepheus [14] is the first cryptographic file system that considers sharing of files and introduces the idea of lazy revocation for improving performance. However, key management in Cepheus is centralized by using a trusted key server for key distribution. More recent cryptographic file systems, such as Oceanstore [23] and Plutus [21], acknowledge the benefit of decentralized key distribution and propose that key management is handled by file owners themselves. For efficient operation, Plutus adopts a lazy revocation model and uses a key-updating scheme based on RSA, as described in Section 4.2.

Farsite [3], SNAD [26] and SiRiUS [15] use public-key cryptography for key management. The group encryption key is encrypted with the public keys of all group members and these lockboxes are stored on the storage servers. This approach simplifies key management, but the key storage per group is proportional to the number of users in the group. Neither of these systems addresses efficient user revocation.

7 Conclusions

Motivated by the practical problem of efficient key management for cryptographic file systems that adopt lazy revocation, we define formally the notion of key-updating schemes for lazy revocation and its security. In addition, we give two methods for additive and multiplicative composition of two secure key-updating scheme into a new scheme which can handle a larger number of user revocations, while preserving security. We also prove the security of two slightly modified existing constructions and propose a new construction, the binary-tree scheme, that balances the tradeoffs of the existing constructions. Finally, we provide a systematic analysis of the computational and communication complexities of the three constructions.

We can extend the definition of key-updating schemes to support user keys for interval t, from which only keys of the time intervals between i and t can be extracted, for any $1 \leq i \leq t$. This is useful in a model in which users joining the group at a later time interval should not have access to past information. The extension can be incorporated in the tree construction without additional cost, but the chaining and trapdoor permutation constructions do not work in this model because the user key reveals all previous keys.

In a companion paper [5], we show how to extend secure key-updating schemes to cryptosystems with lazy revocation, and introduce the notions of symmetric encryption, message-authentication codes, and signature schemes with lazy revocation. Furthermore, we demonstrate that using these cryptosystems in some existing distributed cryptographic file systems improves their efficiency and security.

References

[1] M. Abdalla and M. Bellare, "Increasing the lifetime of a key: A comparitive analysis of the security of rekeying techniques," in *Proc. Asiacrypt 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 546–559, Springer-Verlag, 2000.

[2] M. Abdalla and L. Reyzin, "A new forward-secure digital signature scheme," in *Proc. Asiacrypt 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 116–129, Springer-Verlag, 2000.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, Usenix, 2002.

[4] R. Anderson, "Two remarks on public-key cryptology," Technical Report UCAM-CL-TR-549, University of Cambridge, 2002.

[5] M. Backes, C. Cachin, and A. Oprea, "Lazy revocation in cryptographic file systems," Research Report RZ 3628, IBM Research, Aug. 2005.

[6] M. Bellare and S. Miner, "A forward-secure digital signature scheme," in *Proc. Crypto 1999*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 431–448, Springer-Verlag, 1999.

[7] M. Bellare and B. Yee, "Forward-security in private-key cryptography," in *Proc. CT-RSA 2003*, vol. 2612 of *Lecture Notes in Computer Science*, pp. 1–18, Springer-Verlag, 2003.

[8] M. Blaze, "A cryptographic file system for Unix," in *Proc. First ACM Conference on Computer and Communication Security (CCS)*, pp. 9–16, 1993.

[9] R. Canetti, S. Halevi, and J. Katz, "A forward-secure public-key encryption scheme," in *Proc. Eurocrypt 2003*, vol. 2656 of *Lecture Notes in Computer Science*, pp. 255–271, Springer-Verlag, 2003.

[10] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for Unix," in *Proc. USENIX Annual Technical Conference 2001, Freenix Track*, pp. 199–212, 2001.

[11] Y. Dodis, M. Franklin, J. Katz, A. Miyaji, and M. Yung, "Intrusion-resilient public-key encryption," in *Proc. CT-RSA 2003*, vol. 2612 of *Lecture Notes in Computer Science*, pp. 19–32, Springer-Verlag, 2003.

[12] Y. Dodis, J. Katz, S. Xu, and M. Yung, "Key insulated public-key cryptosystems," in *Proc. Eurocrypt 2002*, vol. 2332 of *Lecture Notes in Computer Science*, pp. 65–82, Springer-Verlag, 2002.

[13] Y. Dodis, J. Katz, and M. Yung, "Strong key-insulated signature schemes," in *Proc. Workshop of Public Key Cryptography (PKC)*, vol. 2567 of *Lecture Notes in Computer Science*, pp. 130–144, Springer-Verlag, 2002.

[14] K. Fu, "Group sharing and random access in cryptographic storage file systems," Master's thesis, Massachusetts Institute of Technology, 1999.

[15] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pp. 131–145, ISOC, 2003.

[16] M. T. Goodrich, J. Z. Sun, and R. Tamassia, "Efficient tree-based revocation in groups of low-state devices," in *Proc. Crypto 2004*, vol. 3152 of *Lecture Notes in Computer Science*, pp. 511–522, Springer-Verlag, 2004.

[17] J. Goshi and R. E. Ladner, "Algorithms for dynamic multicast key distribution trees," in *Proc. 22nd Symposium on Principles of Distributed Computing (PODC)*, pp. 243–251, ACM, 2003.

[18] G. Itkis, "Forward security, adaptive cryptography: Time evolution." Survey, available from `http://www.cs.bu.edu/fac/itkis/pap/forward-secure-survey.pdf`.

[19] G. Itkis and L. Reyzin, "Forward-secure signatures with optimal signing and verifying," in *Proc. Crypto 2001*, vol. 2139 of *Lecture Notes in Computer Science*, pp. 332–354, Springer-Verlag, 2001.

[20] G. Itkis and L. Reyzin, "SiBIR: Signer-base intrusion-resilient signatures," in *Proc. Crypto 2002*, vol. 2442 of *Lecture Notes in Computer Science*, pp. 499–514, Springer-Verlag, 2002.

[21] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[22] H. Krawczyk, "Simple forward-secure signatures from any signature scheme," in *Proc. 7th ACM Conference on Computer and Communication Security (CCS)*, pp. 108–115, 2000.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 190–201, ACM, 2000.

[24] T. Malkin, D. Micciancio, and S. Miner, "Efficient generic forward-secure signatures with an unbounded number of time periods," in *Proc. Eurocrypt 2002*, vol. 2332 of *Lecture Notes in Computer Science*, pp. 400–417, Springer-Verlag, 2002.

[25] T. Malkin, S. Obana, and M. Yung, "The hierarchy of key evolving signatures and a characterization of proxy signatures," in *Proc. Eurocrypt 2004*, vol. 3027 of *Lecture Notes in Computer Science*, pp. 306–322, Springer-Verlag, 2004.

[26] E. Miller, D. Long, W. Freeman, and B. Reed, "Strong security for distributed file systems," in *Proc. the First USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[27] O. Rodeh, K. Birman, and D. Dolev, "Using AVL trees for fault tolerant group key management," *International Journal on Information Security*, vol. 1, no. 2, pp. 84–99, 2001.

[28] A. T. Sherman and D. A. McGrew, "Key establishment in large dynamic groups using one-way function trees," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 444–458, 2003.

[29] R. Tamassia and N. Triandopoulos, "Computational bounds on hierarchical data processing with applications to information security," in *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, 2005.

[30] C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communications using key graphs," *IEEE/ACM Transactions on Networking*, vol. 8, no. 1, pp. 16–30, 2000.