Batch Updates of Key Trees

Xiaozhou Li, Y. Richard Yang,
Mohamed G. Gouda, Simon S. Lam

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188
{*xli, yangyang, gouda, lam*}*@cs.utexas.edu*

TR-2000-22

September 19, 2000

Abstract

[2] introduced the concept of *key trees* for secure group communications and discussed how to update a key tree due to a single rekey request (a join or a leave). In a real application, however, rekey requests are likely to be processed in batches, instead of in real-time. In this paper we consider the case where there are equal number of joins and leaves in a batch. In this case, we can replace a leave by a join at the same location in the key tree. We call this an *update*.

We describe a method, called the *rekey subtree method*, to efficiently process batch updates. The method identifies the keys that need to be updated and updates those keys only once. The method saves server cost substantially, compared to the naive approach, which processes updates one after another. We also describe how to modify the rekey subtree method when joins and leaves are unequal.

To analyze the rekey subtree method's performance, we derive the exact expressions for the best, worst, and average case server's encryption costs. We also derive the exact expression for a user's average decryption cost.

If an application only wants to minimize the server's cost, it can use server cost expressions to decide what degree the key tree should use. Our analysis and experiments show that 4 is usually the optimal degree when the number of updates is below some threshold (about 1/4 of the number of users), when the number of updates is above the threshold, key star (key tree of degree equal to the number of users) is optimal. If the application has users of limited capacity, it should choose a larger degree to reduce the user's work, at the cost of increasing the server's.

1 Introduction

With the growth of the Internet, there is a need for secure multicast applications. [2] introduced the concept of *key graphs* for secure group communications. Among all kinds of key graphs, *key tree* and *key star* are two particularly interesting structures. A key star can be considered as a special key tree, with degree equal to the number of users. Suppose there are n users, for a key tree with degree d and height h (so h is about $\log_d n$), the server's encryption cost (simply called *server cost*) is $2(h-1)$ for a join, $d(h-1)$ for a leave. It can be shown that $d = 4$ minimizes server cost.

In a real application, however, joins and leaves are likely to be processed periodically, instead of in real-time. The server collects join/leave requests for a period and processes them in a batch. To simplify discussion, we make the following two assumptions in this paper:

1. A single key server handles all the rekey requests.
2. There are equal number of joins and leaves in a batch.

A real application may distribute the server's work among a number of entities. In that case, the server cost notion in this paper should be considered as server side cost. For many secure multicast applications, the group size is unlikely to drastically change, so we can assume that there are equal number of joins and leaves in a batch. Our results are still useful when joins and leaves are roughly equal.

It is more efficient to replace a leaving user by a joining user, because it reduces the number of keys to be updated. We call a leave and a join at the same location in the key tree an *update*.

We use m to denote the number of updates, n the number of users, d the degree of the key tree, h the height of the key tree. The naive approach, which processes updates one after another, is not scalable. Each update's cost is $d(h-1)$, so the cost for m updates is $m \cdot d(h-1)$. The naive approach is inefficient because it updates some keys multiple times, while those keys only need to be updated once.

We describe a method, called *rekey subtree method*, to efficiently process batch updates. The method uses a simple algorithm to identify the *rekey subtree*, which contains only the keys that need to be updated, and updates those keys only once. Compared to the naive approach, the rekey subtree method saves server cost substantially. We also discuss how to modify the rekey subtree method when joins and leaves are unequal.

To analyze the rekey subtree method's performance, we derive the exact expressions for the best case, worst case, and average case server costs. We also derive a user's average decryption cost (simply called *average user cost*.

If an application only wants to minimize server cost, it can use these expressions to decide what degree the key tree should use. Our analysis and experiments show that $d = 4$ is usually optimal when the number of updates is below some threshold, when the number of updates is above the threshold,

$d = n$ (key star) is optimal. We show how to compute this threshold. The threshold is roughly $n/4$. If the application has users of limited capacity, the application should choose a larger d to reduce the user's work, at the cost of increasing the server's.

The paper is organized as follows. Section 2 describes the rekey subtree method. Section 3 derives the exact expressions for the server cost and user cost. Section 4 discusses how to use these expressions to minimize server cost. Section 5 discusses related work. Section 6 concludes the paper.

2 Rekey subtree method

2.1 Notations

We use the following notations: m is the number of updates, n the number of users, d degree of the key tree, S the size (number of nodes) of the rekey subtree.

2.2 The idea

When there is a single update, all the keys on the path from the update location to the root of the key tree have to be updated. When there are multiple updates, there are multiple paths. These paths form a subtree. Keys located on multiple paths only need to be updated once, instead of multiple times. If we can identify this subtree (called *rekey subtree*) and update the keys in the rekey subtree only once, we can reduce the server cost.

Figure 1 shows a small example to illustrate this idea. In this example, u_1 is replaced by u'_1, u_4 is replaced by u'_4. The rekey subtree consists of three nodes: k'_{1-9}, k'_{123}, and k'_{456}. Note that k'_1 and k'_4 are not in the rekey subtree because they are individual keys and are established separately. In this example, k_{1-9} only need to be updated once, not twice.

If we use key-oriented or group oriented rekeying, and the rekey subtree's size is S, then the server cost is $d \cdot S$. S is usually smaller than $m \log_d n$, the sum of the lengths of the rekey paths. The difference depends on the exact locations of the updates. We quantify this difference in Section 3.

2.3 How to identify the rekey subtree

The algorithm to identify the rekey subtree is simple. We are given a number of updates. For each update, we mark all the ancestors of the update in the key tree by following the parent pointer of a node. During the process, if we find an ancestor that has been marked, we proceed to process the next update. The resulting marked nodes form the rekey subtree. The running time of the algorithm is upper bounded by $m \log_d n$. The exact running time depends on the update locations. Note that the marking process is not a performance burden because it requires little computation.

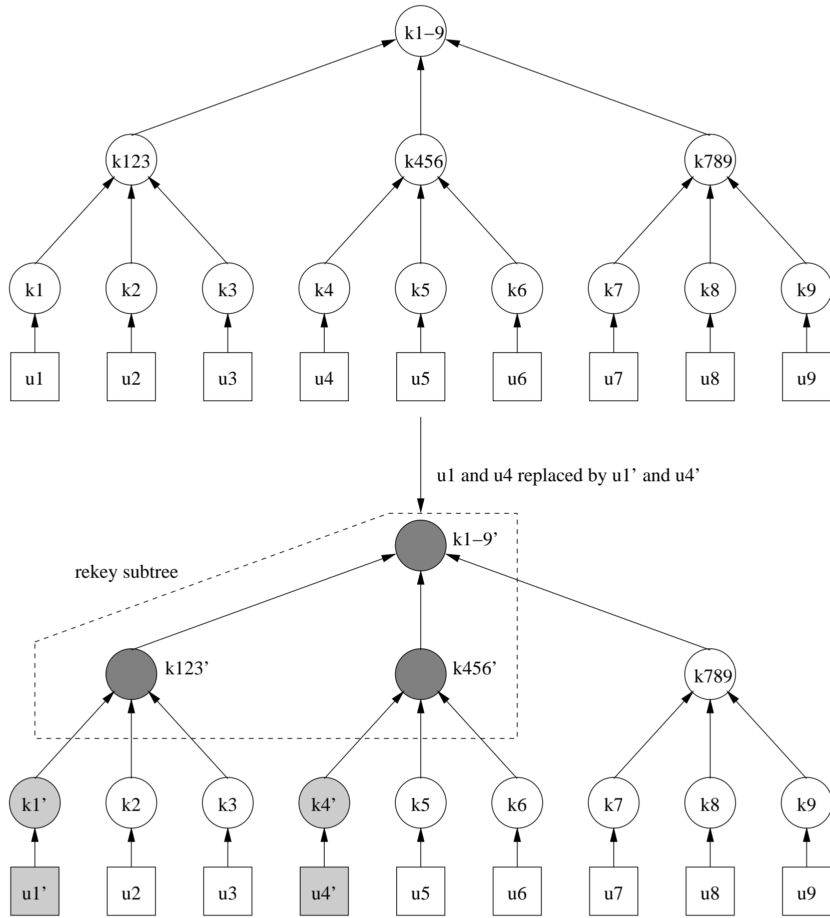


Figure 1: Example of a rekey subtree

2.4 Batch update protocol

Once we have identified the rekey subtree, rekeying can be done efficiently. We discuss the protocols for three rekeying strategies: user-oriented, key-oriented, and group-oriented ([2]).

2.4.1 User-oriented rekeying

In this approach, the server constructs a rekey message that contains precisely the new keys needed by the user and encrypts them using a key held by the user. For the example in Figure 1, key server s needs to send the following rekey messages:

$s \rightarrow \{u_7, u_8, u_9\}$: $\{K'_{1-9}\}_{K_{789}}$
$s \rightarrow \{u'_1\}$: $\{K'_{1-9}, K'_{123}\}_{K'_1}$
$s \rightarrow \{u_2\}$: $\{K'_{1-9}, K'_{123}\}_{K_2}$
$s \rightarrow \{u_3\}$: $\{K'_{1-9}, K'_{123}\}_{K_3}$
$s \rightarrow \{u'_4\}$: $\{K'_{1-9}, K'_{456}\}_{K'_4}$
$s \rightarrow \{u_5\}$: $\{K'_{1-9}, K'_{456}\}_{K_5}$
$s \rightarrow \{u_6\}$: $\{K'_{1-9}, K'_{456}\}_{K_6}$

The rekey messages can be constructed as follows. For each node x in the rekey subtree, and for each unchanged child y of x, the server constructs a rekey message by encrypting the new keys of k-node x and all its ancestors (up to the root) by the key K_y of y. This rekey message is then multicasted to userset(K).

In terms of server cost, user-oriented rekeying is not as efficient as key-oriented or group-oriented rekeying, because it encrypts the same key using many descendents' keys. For the above example, K'_{1-9} are encrypted by 7 different keys, while in other rekeying strategies, encrypting K'_{1-9} by 3 different keys, K'_{123}, K'_{456}, K_{789}, is sufficient.

2.4.2 Key-oriented rekeying

In this approach, each new key is encrypted individually. The protocol is described in Figure 2. For the example in Figure 1, the server sends out the following rekey messages:

$s \rightarrow \{u_7, u_8, u_9\}$: $\{K'_{1-9}\}_{K_{789}}$
$s \rightarrow \{u'_1\}$: $\{K'_{1-9}\}_{K'_{123}}, \{K'_{123}\}_{K'_1}$
$s \rightarrow \{u_2\}$: $\{K'_{1-9}\}_{K'_{123}}, \{K'_{123}\}_{K_2}$
$s \rightarrow \{u_3\}$: $\{K'_{1-9}\}_{K'_{123}}, \{K'_{123}\}_{K_3}$
$s \rightarrow \{u'_4\}$: $\{K'_{1-9}\}_{K'_{456}}, \{K'_{456}\}_{K'_4}$
$s \rightarrow \{u_5\}$: $\{K'_{1-9}\}_{K'_{456}}, \{K'_{456}\}_{K_5}$
$s \rightarrow \{u_6\}$: $\{K'_{1-9}\}_{K'_{456}}, \{K'_{456}\}_{K_6}$

s traverses the rekey subtree in pre-order
  let x_j be the currently visited node,
    $x_0, \ldots, x_j$ be the path from the root to x_j,
    $K'_0, \ldots, K'_j$ be the new keys for these nodes.
  for each child y of x_j and y not in the rekey subtree
    let K_y be the key for y,
    let $M = \{K'_0\}_{K'_1}, \ldots, \{K'_j\}_{K_y}$
    $s \rightarrow userset(K_y) : M$

Figure 2: Key-oriented batch update protocol

The server cost in this approach is exactly $d \cdot S$.

2.4.3 Group-oriented rekeying

In this approach, the server constructs a single rekey message containing all the new keys. This rekey message is then multicasted to the entire group. The way to encrypt the new keys is similar to key-oriented rekeying. For the example in Figure 1, s does the following:

$$
\begin{aligned}
&s \rightarrow \{u_1', u_2, u_3, u_4', u_5, \ldots, u_9\}: \\
&\quad \{K_{1-9}'\}_{K_{123}'}, \{K_{1-9}'\}_{K_{456}'}, \{K_{1-9}'\}_{K_{789}} \\
&\quad \{K_{123}'\}_{K_1'}, \{K_{123}'\}_{K_2}, \{K_{123}'\}_{K_3}, \{K_{456}'\}_{K_4'}, \{K_{456}'\}_{K_5}, \{K_{456}'\}_{K_6}
\end{aligned}
$$

The server cost is $d \cdot S$, same as key-oriented rekeying.

2.5 Modifications for more joins than leaves

If there are more joins than leaves, the batch can be considered as some updates plus by some joins. For the extra joins, the server creates a subtree and find a location in the key tree to graft the subtree. Then the joins can be considered as updates too. Figure 3 shows a small example to explain the idea. In this example, the rekey subtree consists of three nodes: K_{1-9}', K_{456}', and K_{789}.

2.6 Modifications for more leaves than joins

If there are more leaves than joins, it is possible that some parts of the key tree will be pruned out. In this case, it is necessary to find out which keys need to be updated, and which need to be removed from the key tree. This is easy to do: if all the children of a key are removed, than this key should be removed too. We can easily add this feature to the marking process. Figure 4 shows a small example. In this example, the rekey subtree consists of two nodes: K_{1-9}', K_{123}'. K_{456}' does not belong to the rekey subtree because all its children are removed.

3 Analysis

In Section 2, we have seen that the rekey subtree method's cost depends on the exact locations of the updates. If the updates are concentrated, the rekey subtree is smaller, if the updates are scattered, the rekey subtree is bigger. In this section, we derive the exact expressions for the best, worst, and average case server costs and user average cost in terms of m, n, and d.

3.1 Assumptions

We make the following assumptions to simplify the discussion.

1. n is some power of d, namely, the key tree is a complete d-ary tree, with n leaves at the bottom. We call the leaf locations location 1, ..., location n.

2. Every receiver has an equal probability of being updated.

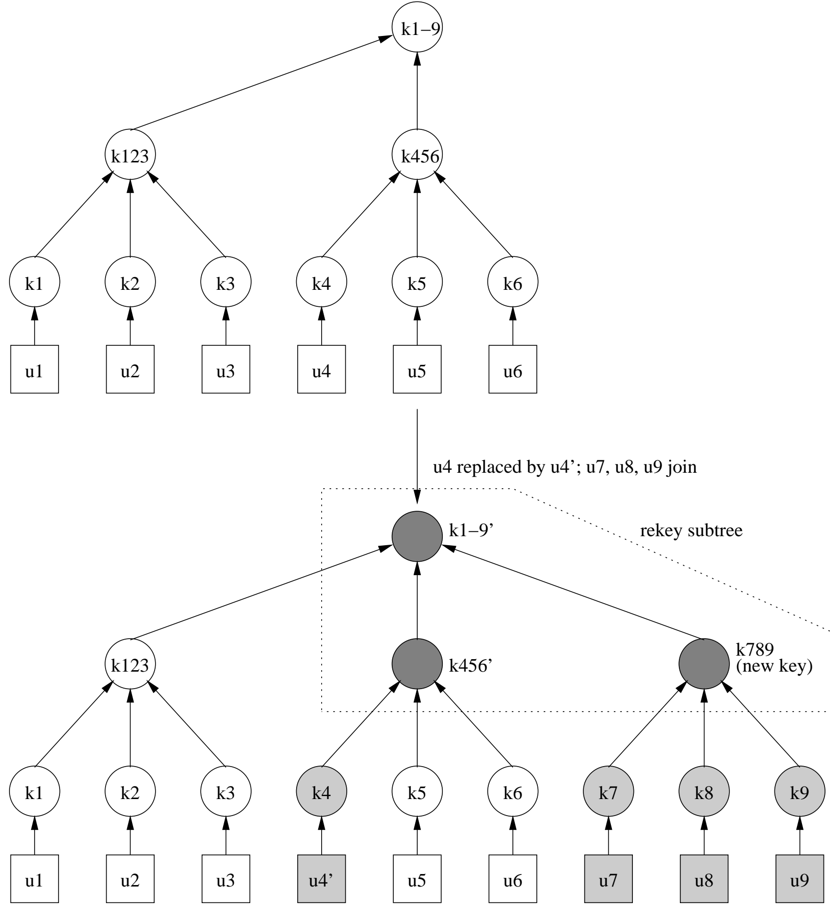


Figure 3: Example of more joins than leaves

3.2 Best case server cost

Intuitively, if all the updates happen at concentrated locations, the rekey subtree is smaller. It can be proved that the best case happens when the updates are at location 1, ..., location m, as shown in Figure 5. The proof is omitted for brevity. It is not hard to derive the size of the rekey subtree in this case. It is:

$$S_{best} = \begin{cases} \frac{1}{d-1}(m - digit_sum(m,d)) + \sum_{i=1}^{k} rem(m, d^i) + (h - k + 1) & 1 \leq m < n \\ \frac{n-1}{d-1} & m = n \end{cases}$$

where $k = \lfloor \log_d m \rfloor$, $h = \log_d n + 1$, $digit_sum(m,d)$ is the sum of the digits of m when written in radix d, $rem(a,b)$ is a function that has value 1 if $a \bmod b \neq 0$, value 0 if $a \bmod b = 0$.

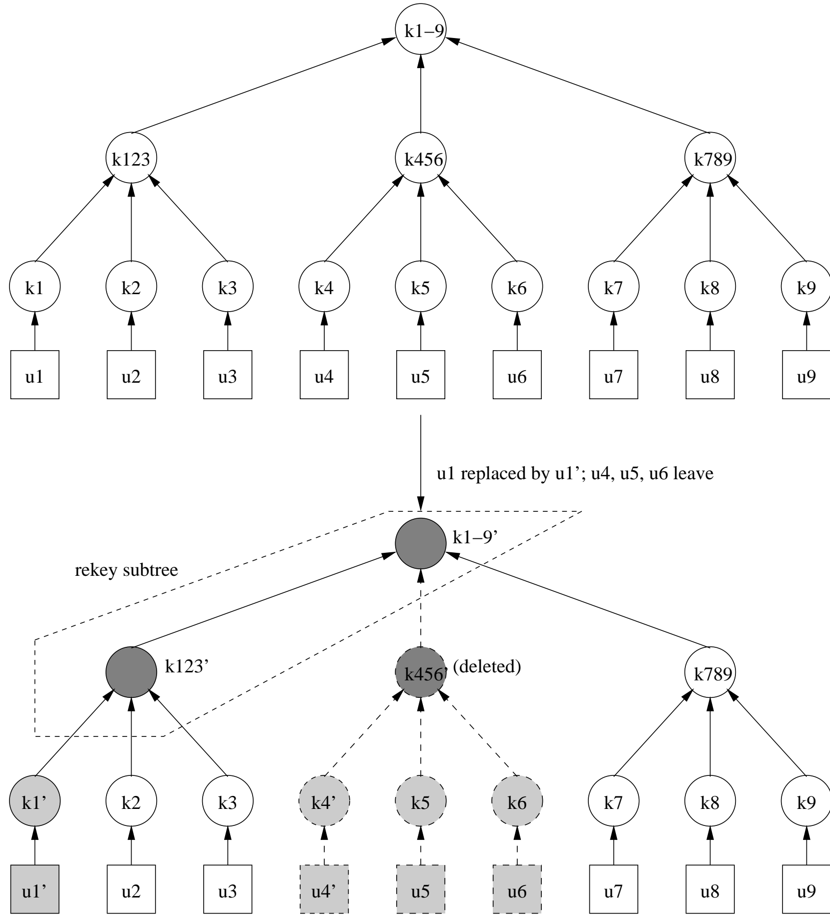


Figure 4: Example of more leaves than joins

The best server cost, $cost_{best}$, is thus $d \cdot S_{best}$.

3.3 Worst case server cost

Intuitively, if the m updates are uniformly distributed among locations 1 to n, the rekey subtree's size is maximized. Suppose $d^k \leq m < d^{k+1}$ and $m = r \cdot d^k + q$, $1 \leq r \leq d-1$, $0 \leq q < d^k$. Figure 6 shows one of the worst case scenarios. Basically, the worst case scenario happens if we distribute the m updates evenly on subtrees $T_1, \ldots, T_{d^k}$. It can be proved that this scenario really maximizes the rekey subtree. The proof is omitted for brevity. It is not hard to derive the worst rekey subtree size:

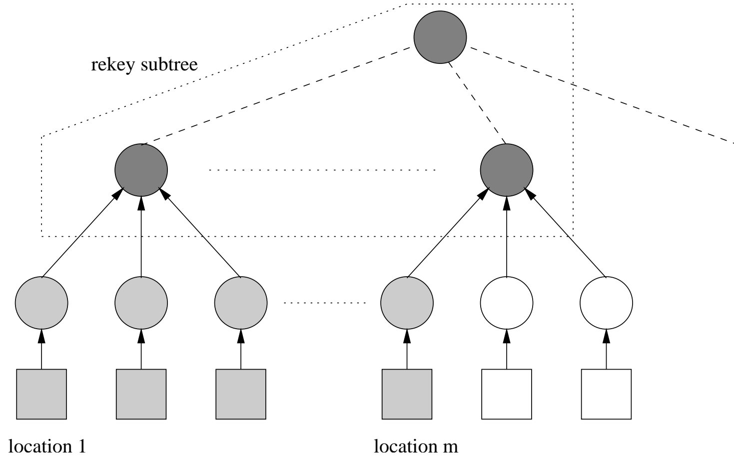


Figure 5: Best case scenario.

$$S_{worst} = \begin{cases} m \cdot (h-k-2) + \frac{1}{d-1}(d^{k+1}-1) & 1 \leq m < n \\ \frac{n-1}{d-1} & m = n \end{cases}$$

where $k = \lfloor \log_d m \rfloor$, $h = \log_d n + 1$.

The worst server cost, $cost_{worst}$, is thus $d \cdot S_{worst}$.

3.4 Average case server cost

What is the rekey subtree's average size, given m, n, d? To answer this question, we first consider the rekey subtree's size on a certain level. Summing up the sizes on all levels will give the size of the rekey subtree.

First note that each node on the same level is equivalent. So if we can find out the probability that a node is in the rekey subtree, then we can find out the rekey subtree's size on that level, by multiplying the probability with the number of nodes on that level.

We use the following convention: the root is on level 0, leaves are on level $h-1$, where $h = \log_d n+1$. Consider a node x on level l, $0 \leq l \leq h-2$. (We don't need to consider keys on level $h-1$ because they are individual keys and don't belong to the rekey subtree.) The subtree rooted at node x has $n' = d^{h-l-1}$ leaves. If one of these leaves is updated, then x is in the rekey subtree, otherwise x is not in the rekey subtree. The probability that none of the m updates is located in the subtree of x is:

$$Pr(l) = \frac{C_{n-n'}^m}{C_n^m} = \frac{(n-n')(n-n'-1)\ldots(n-n'-m+1)}{n(n-1)\ldots(n-m+1)},$$

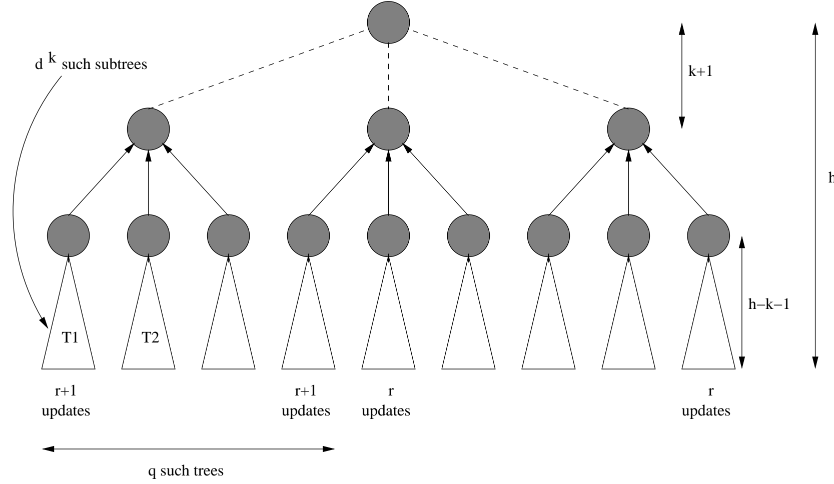


Figure 6: Worst case scenario.

where C_a^b is the number of ways to pick b elements out of a elements, regardless of order. We use the convention that $C_a^b = 0$ if $b > a$. We note that $Pr(l)$ is easy to compute, even and m and n are large.

Thus, the rekey subtree's size on level l is $d^l(1 - Pr(l))$. Summing up over all the levels, we obtain the average size of the rekey subtree:

$$S_{avg} = \sum_{l=0}^{h-2} d^l (1 - \frac{C_{n-n/d^l}^m}{C_n^m}) = \frac{n-1}{d-1} - \sum_{l=0}^{h-2} \frac{C_{n-n/d^l}^m}{C_n^m}.$$

The average cost, $cost_{avg}$, is thus $d \cdot S_{avg}$.

3.5 Shapes of the costs and validation of analysis

The properties of the three costs are not obvious from their expressions. Figure 8 shows the shapes of these costs. We also implemented a simulation program to validate our analysis for the average cost. The simulation program randomly generates updates and computes the rekey subtree size accordingly. Our analysis matches with the simulation perfectly.

Basically, $cost_{best}$ grows linearly with m. $cost_{worst}$ reaches its maximum, $\frac{d}{d-1}(n-1)$, when $m = n/d$. Intuitively, when uniformly distributed, n/d updates can make every non-leaf node in the key tree belong to the rekey subtree. $cost_{avg}$ is closer to $cost_{worst}$ than to $cost_{best}$.

3.6 Rekey subtree vs. naive approach

Since the rekey subtree method only updates keys once. It outperforms the naive approach even when there are only two updates. The difference becomes

more substantial as the number of updates grows. Figure 9 compares the rekey subtree method's worst cost and the naive approach's cost. We observe that even the rekey subtree's worst cost is much better than the naive approach.

3.7 User average cost

Intuitively, a user's average cost should decrease as d grows, should increase as m grows. It can be derived as follows. Figure 7 illustrates the idea. Denote h as the height of the key tree ($h = \log_d n + 1$), $T(x)$ the subtree rooted at node x, $L(x)$ the set of leaves in $T(x)$, $|L(x)|$ the size of $L(x)$. User u decrypts $i(1 \leq i \leq h-1)$ keys when there is no update in $L(y)$ and there is at least one update in $L(x)$. Since $|L(y)| = d^{h-i-1}$, $|L(x)| = d^{h-i}$,

$$Pr(u \text{ decrypts } i \text{ keys}) = (C^m_{n-|L(y)|} - C^m_{n-|L(x)|})/C^m_n$$

where C^b_a has the same definition as in Section 3.4. When $i = h$, u decrypts i keys when either itself or any of its $(d-1)$ siblings is updated, thus

$$Pr(u \text{ decrypts } h \text{ keys}) = 1 - C^m_{n-d}/C^m_n$$

Summing them up, we have the user's average decryption cost:

$$user_{avg} = \sum_{i=1}^{h-2} \frac{i(C^m_{n-n/d^i} - C^m_{n-n/d^{i-1}})}{C^m_n} + \frac{(h-1)(C^m_n - C^m_{n-d})}{C^m_n}$$

Figure 12 shows the shape of this function.

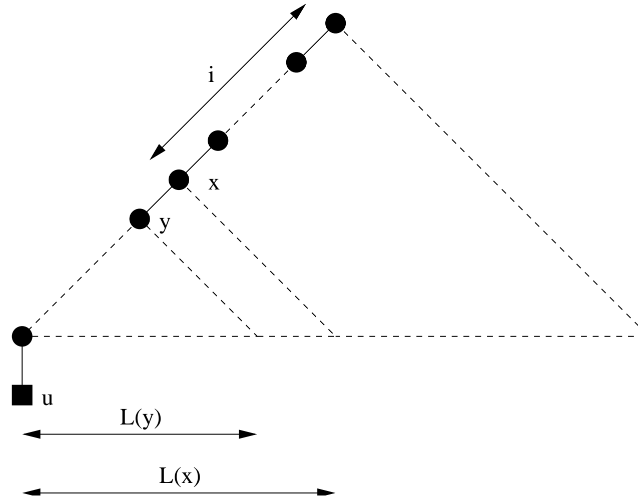


Figure 7: User's average decryption cost.

4 Minimization of server cost

A real application usually can estimate the expected m and n, then it would like to know which d it should choose to minimize server cost. We consider the average cost. Since the average cost is expressed in terms of m, n, d, the application can plug in the average case expression to compare the costs for different d's. Figure 10 and 11 compares the costs for different d's for a wide variety of n's. We observe $d = 4$ outperforms other choices when m is not big. (Since the expression is available, it is possible to find the theoretical optimal d. This might be hard since the expression is not simple.)

We also observe that if we choose $d = 4$, when m is above some threshold (denoted by $\bar{m}$), the average cost is above n. Recall that if we use a key star, the server cost is always n. So if an application estimates that m is likely to be above $\bar{m}$, it might want to use key star.

We can make $cost_{avg} = n$, plug in $d = 4$, and compute $\bar{m}$. Since $cost_{avg}$ is a complicated expression, solving this equation mathematically may not be feasible. However, we observe $cost_{avg}$ is a monotonic function, so we can use binary search to locate $\bar{m}$. Table 1 shows some $\bar{m}$ values. We observe that $\bar{m}/n$ is about 0.25. So the general rule for an application to minimize server cost is: if m is less than $n/4$, choose $d = 4$, otherwise choose $d = n$, namely, use a key star.

n	$\bar{m}$	$\bar{m}/n$
64	15	0.23
256	63	0.25
1024	247	0.24
4096	991	0.24
16384	3963	0.24
65536	15851	0.24

Table 1: $\bar{m}$ for $d = 4$.

However, if the application has users of limited capacity, then it should consider choosing a larger d to reduce the user's work, at the cost of increasing the server's.

5 Related work

Many rekey methods have been proposed for secure group communications. Many of them are designed to minimize single rekey cost. [1] discusses batch processing for their key management scheme. They used boolean function minimization techniques to reduce the batch rekeying cost. Their approach, however, has the collusion problem, namely, two users can combine their knowledge of auxiliary keys to continue to read group communications, even after they leave the group. We consider collusion a severe problem.

The implementation of [2] can be found in [3]. The interested reader is encouraged to read [3] and [4] for the design and implementation of a group key management system.

6 Conclusion

In this paper, we proposed the rekey subtree method to process batch updates. We described an algorithm to identify the rekey subtree, and a protocol to process batch updates. We also analyzed the rekey subtree's performance (in terms of server cost), under best, worst, and average cases. The rekey subtree method performs much better than the naive approach. We also analyzed a user's average decryption cost.

Using our analytical results, an application can choose an appropriate key tree degree to minimize server cost, or to balance the work between server and user. In general, if an application only wants to minimize the server cost, it should choose degree 4 if the number of updates is below 1/4 of the number of users, otherwise, the application should choose degree equal to number of users, namely, use key star. If some users have limited capability, the application should choose some larger degree to reduce the user's work, at the cost of increasing the server's.

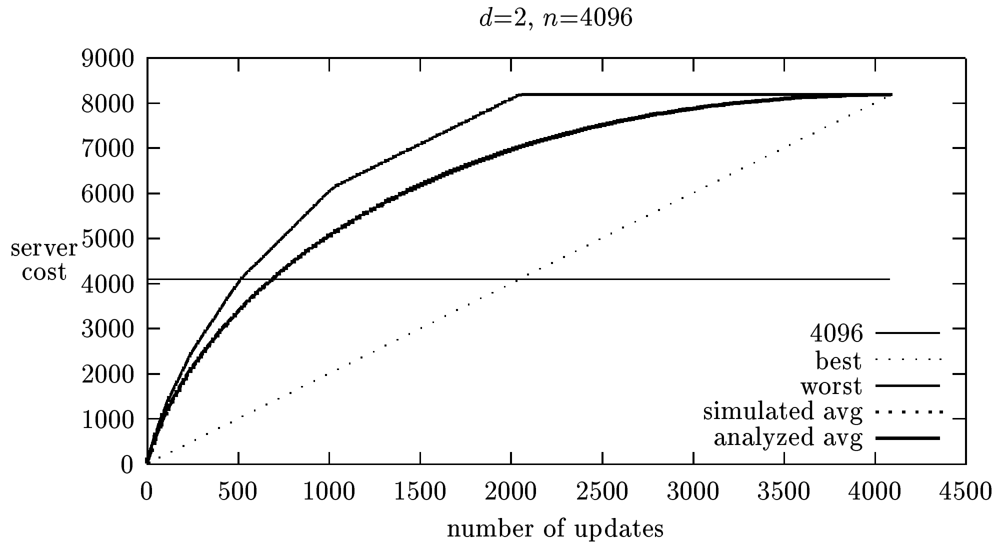
References

[1] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha. Key management for secure internet multicast using boolean function minimization techniques. In *Proceedings of Infocom 99*, 1999.

[2] C. K. Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of Sigcomm 98*, 1998.

[3] C. K. Wong and Simon S. Lam. Keystone: A group key management service. In *Proceedings of the International Conference on Telecommunications*, 2000.

[4] Y. Richard Yang, Min S. Kim, Xincheng Zhang, and Simon S. Lam. Keygem: Towards a scalable and reliable group key management service. Work in progress.



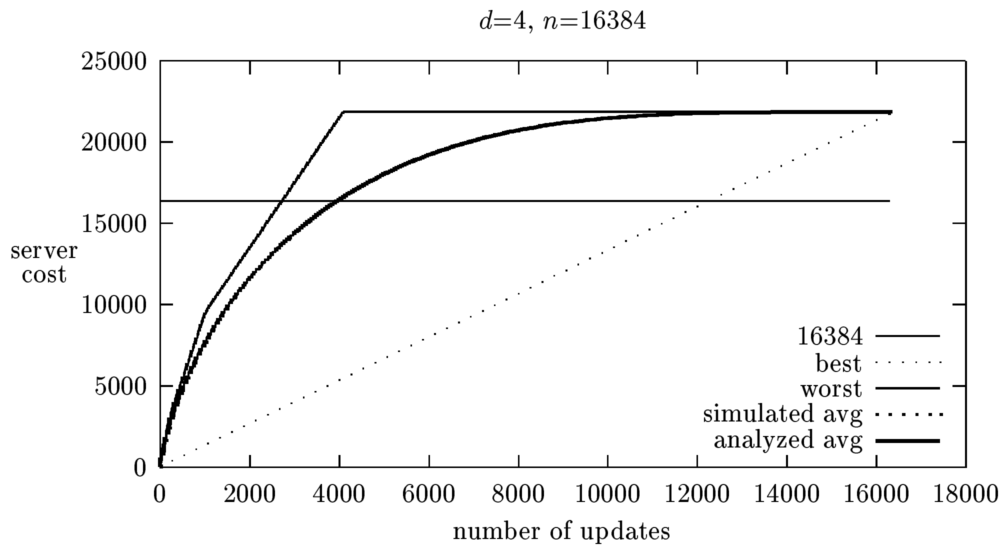
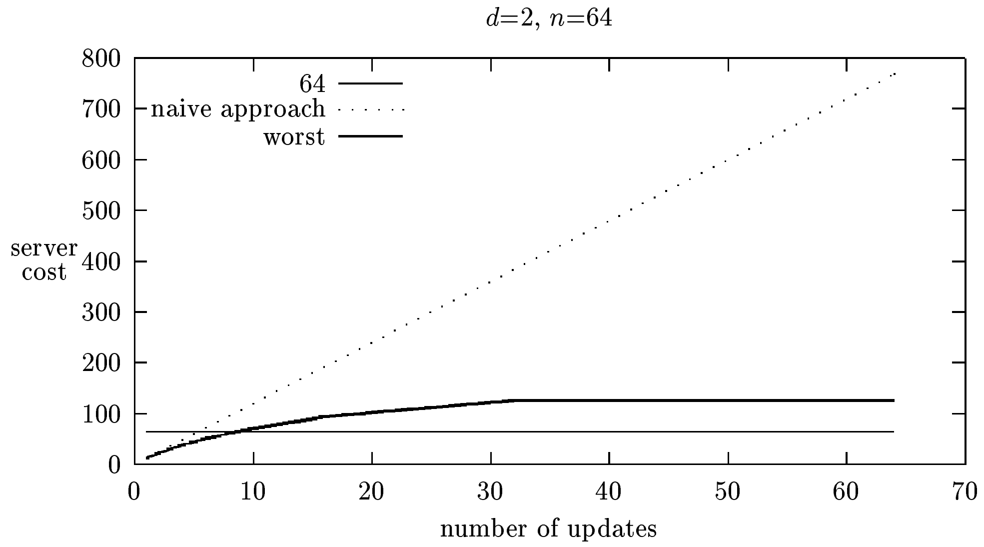


Figure 8: Best, worst, and average case server costs.



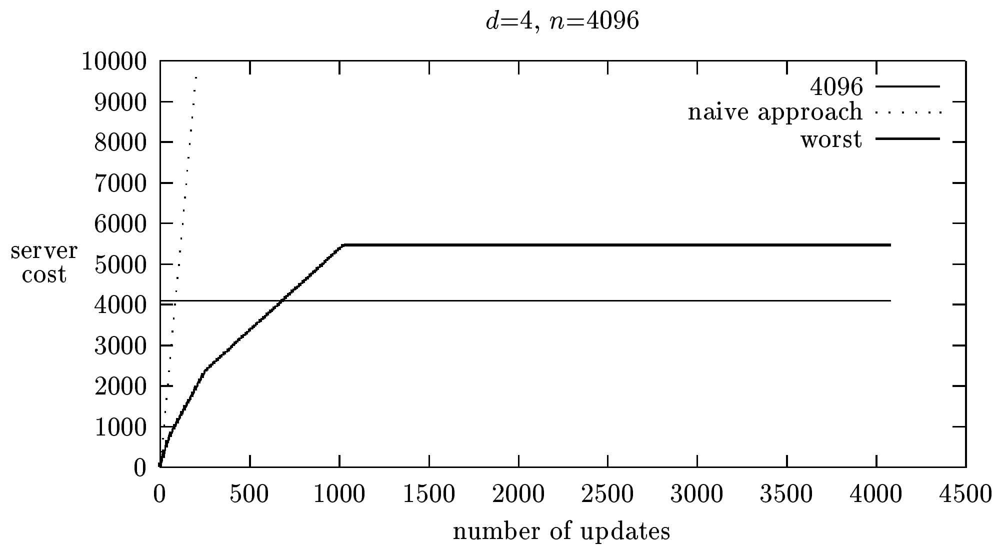


Figure 9: Performance comparison between rekey subtree and naive approach

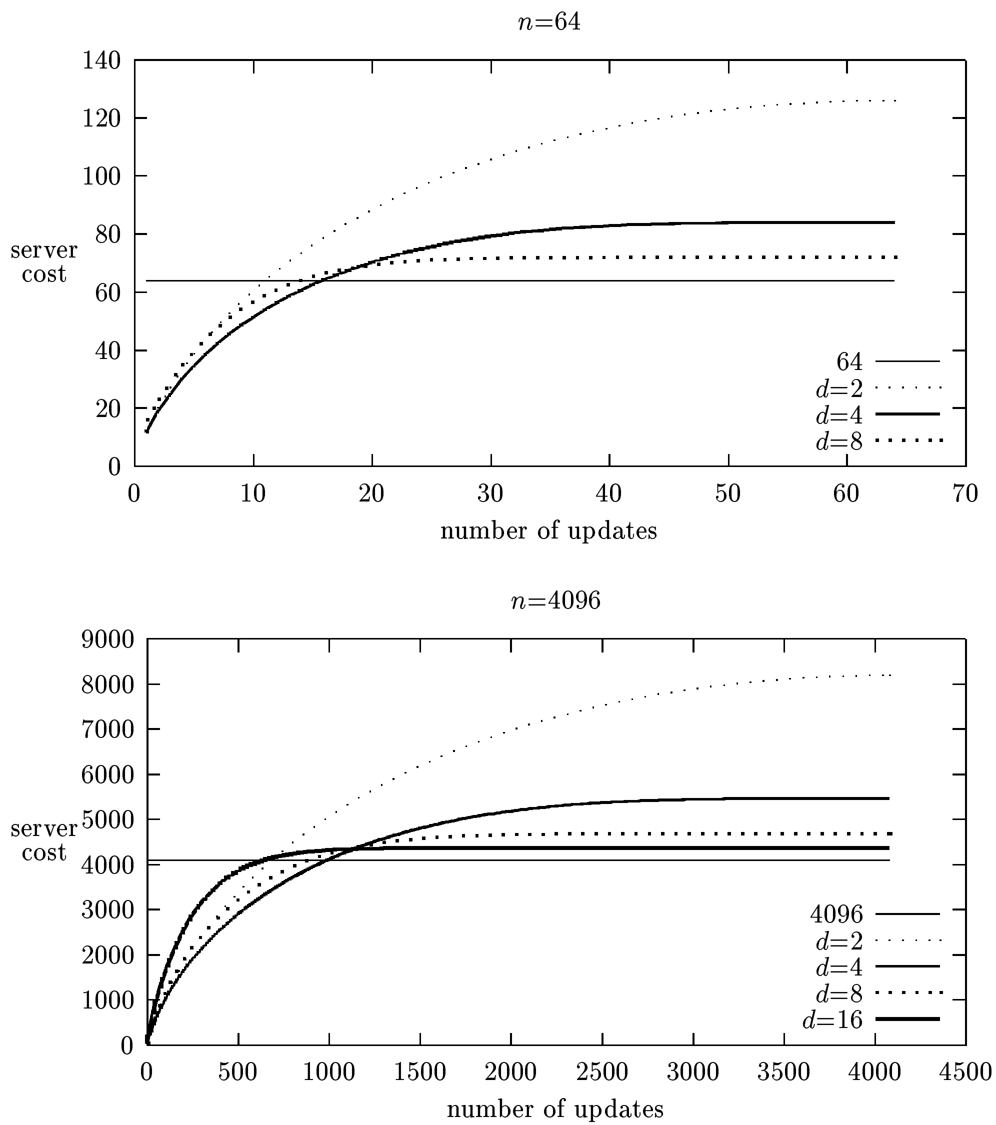
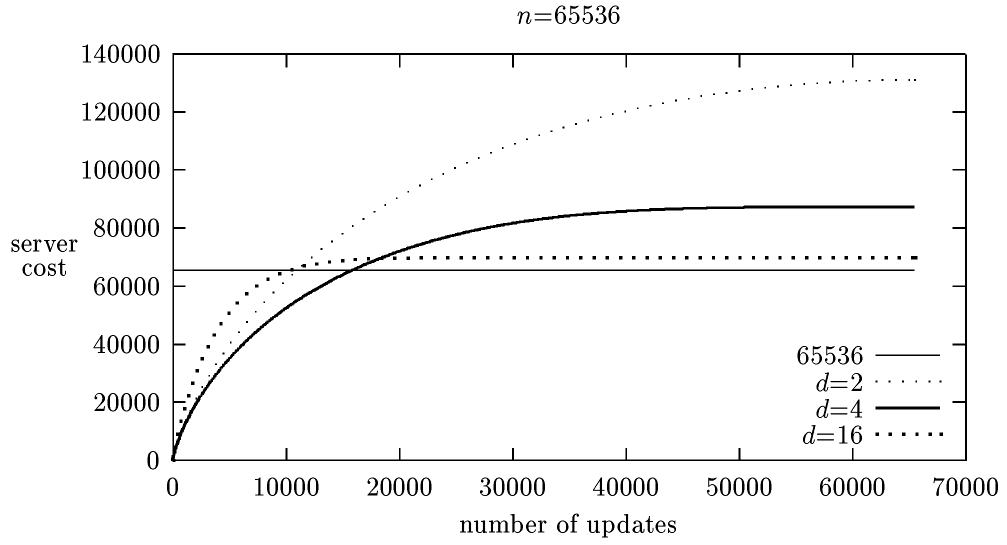


Figure 10: Average case comparison between different d's, on small n's.



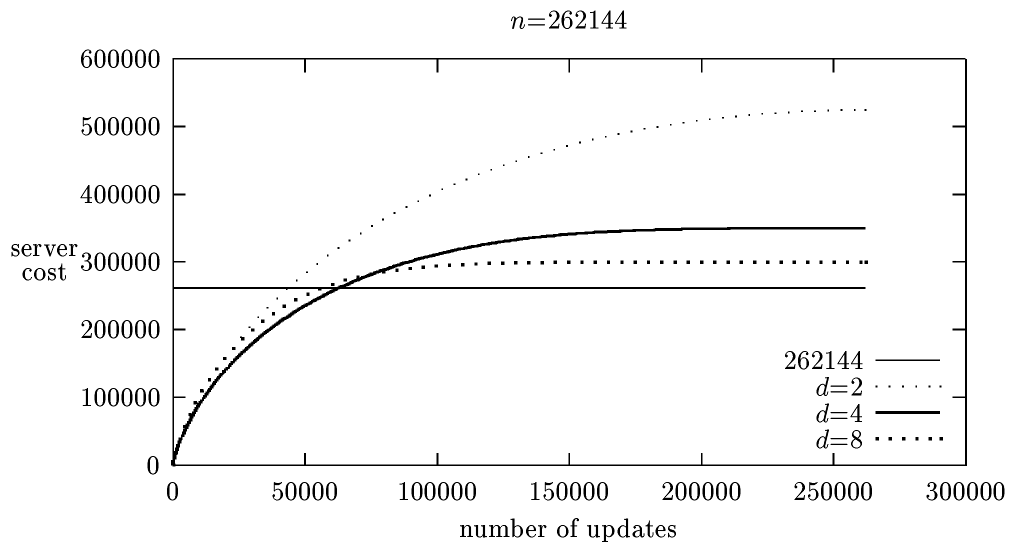


Figure 11: Average case comparison between different d's, on large n's.

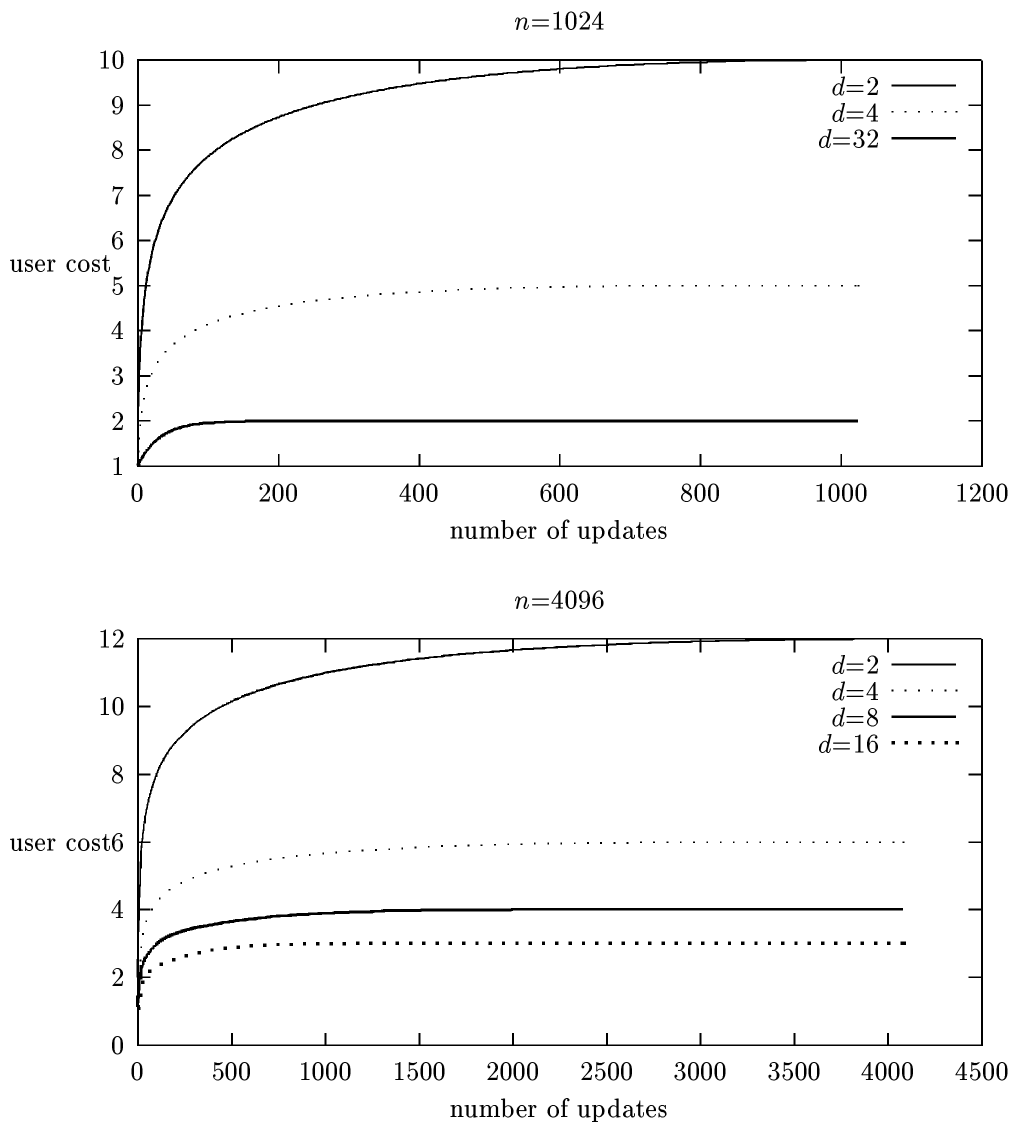


Figure 12: User's average decryption cost.