On the Encipherment of Search Trees and Random Access Files

R. BAYER AND J. K. METZGER

Institut für Informatik der Technischen Universität München, West Germany

The securing of information in indexed, random access files by means of privacy transformations must be considered as a problem distinct from that for sequential files. Not only must processing overhead due to encrypting be considered, but also threats to encipherment arising from updating and the file structure itself must be countered. A general encipherment scheme is proposed for files maintained in a paged structure in secondary storage. This is applied to the encipherment of indexes organized as *B*-trees; a *B*-tree is a particular type of multiway search tree. Threats to the encipherment of *B*-trees, especially relating to updating, are examined, and countermeasures are proposed for each. In addition, the effect of encipherment on file access and update, on paging mechanisms, and on files related to the enciphered index are discussed. Many of the concepts presented may be readily transferred to other forms of multiway index trees and to binary search trees.

Key Words and Phrases: *B*-trees, cryptography, encipherment, indexed sequential files, indexes, paging, privacy, privacy transformation, protection, random access files, search trees, security.
CR Categories: 2.12, 3.73, 3.74, 4.33

1. INTRODUCTION

Cryptology has been traditionally concerned with the concealment of information in humanly readable text strings. Applications to computer processed information have stayed within this tradition by ciphering only sequential strings of data, either those stored in sequential files or those transmitted over communications lines.

That this restriction is no longer practicable is evidenced by the increasing use of sophisticated file structures to maintain valuable databases. Not only are sequential encrypting schemes not feasible for these structures, but new possibilities for cipher "breaking" arise from updating operations as well as from the file structures themselves.

This paper is, accordingly, concerned with developing secure encipherment schemes for information stored in large, randomly accessed, dynamically changing

The research reported in this paper was supported by the Sonderforschungsbereich 49—Elektronische Rechenanlagen und Informationsverarbeitung—of the Deutsche Forschungsgemeinschaft. A version of this paper was presented at the International Conference on Very Large Data Bases, Framingham, Mass., Sept. 22–24, 1975.
Authors' present addresses: R. Bayer, Technical University of Munich, Munich, West Germany (until August 1976, visiting at IBM—K55/282, Monterey and Cottle Roads, San Jose, CA 95193); J.K. Metzger, 45 Delham Ave., Buffalo, NY 14216.

files. Such files will often contain information which is sensitive to disclosure. If they are maintained on storage devices in a computer system accessible to a number of persons, whose actions cannot be completely controlled, then some individual may obtain unauthorized ability to examine the contents of the storage devices, hence compromising security.

This danger may be countered by storing file contents in an enciphered form (*cipher text*) obtained by performing a privacy transformation on the original information (*plain text*). Only if an intruder has information with which to decipher the encrypted text can he obtain sensible information. An additional benefit is that undetectable adulteration of a file is made difficult since falsified data must be properly enciphered.

Most such files will be in the form of an ordered index, organized for external searching. An *index* is a collection of index elements, which are pairs (x, α), where x is a name which is unique to each element, and α is some associated information. α may be "self-contained" or it may contain a pointer to a set of records in a related document file. The basic operations on an index are the insertion and deletion of index elements, and the retrieval of associated information according to name, with the possibility that this information may be updated and rewritten.

Such files will normally reside on secondary storage devices in encrypted form, and only small portions will be in main memory as plain text at any time. Indexes are commonly structured for external searching as multiway trees. The encipherment schemes developed in this paper will be presented in terms of the B-tree type of multiway trees invented by Bayer and McCreight [1]. Many of the ideas, however, may be readily carried over to other multiway tree structures.

The encrypting of index files is not straightforward as in the case of sequentially processed files [2, 4]. Encipherment schemes must take into account the need randomly to access and, more importantly, randomly to alter portions of the file. One must also be concerned with whether successive versions of the index (i.e. after updates) can provide clues which a cryptanalyst might use to break the encipherment.

2. PRIVACY TRANSFORMATIONS

A privacy transformation or *cipher* is a normally reversible function for converting plain text strings to cipher text strings under control of a so-called *cipher key*. If T is the cipher function, k an appropriate cipher key, and A a plain text string, then $T(A, k)$ results in B, the cipher text image of A. If T has an inverse, T', then $T'(B, k)$ recovers the plain text A. In cases where no—or no effectively computable—inverse exists, T is called an *irreversible* or a *one-way cipher* [9, 11]. These are useful precisely when recovery of plain text is to be prevented; novel use will be suggested in Section 6.

Following precepts first laid down by Kerckhoffs in 1883, one should assume that a cryptanalyst may know the general transformation method being used [9] (e.g. in a computing application, he may have bribed, or be, the programmer). Accordingly, the security of a cipher depends on restricting knowledge of which keys were used to encrypt which information objects.

An analyst breaks a cipher by reconstructing plain text from cipher text, either directly or by deducing the key used. This process can be performed remarkably

efficiently for many of the simpler ciphers with the aid of a computer. Tuckerman found, for example, that the cost of breaking a number of schemes ranges from a few hundred to a few thousand dollars [9]. The effort required by a cryptanalyst to break a particular cipher is called its *work factor*, but unfortunately it is not readily quantified. Experience seems to show that high work factors are only obtained by ciphers with high encipherment costs (but the converse is not true) [7].

Breaking a cipher is greatly simplified if it is known that a specific segment of plain text has been enciphered, even if it is not known where it appears in the text. This presents problems for computerized files. One can be careful not to encipher standardized information (e.g. for formatting or padding), but one may not be able to avoid encrypting records which have been planted in the file by the analyst or a collaborator. The larger and more comprehensive a file is, the better the chance of placing a legitimate record with known information in the file. When cleverly selected, this information may permit the analyst to break the encipherment. We shall refer to this as the *planted record problem*.

Computationally, transformations usually operate on fixed size segments of text, e.g. w-bit segments, where w is a convenient number like the machine word size. Let $S = s_1 \ldots s_n$ be a text string divided into w-bit segments, let T be a cipher, and let k_0 be a cipher key for T.[1] Then T normally distributes over S as follows:

$$T(S, k_0) = T(s_1, k_1) \ldots T(s_n, k_n),$$

where the *segment key* k_i for text segment s_i is calculated by a relation of the form $k_i = f(k_0, i)$. Two important classes of ciphers are the following:

(a) T is a *block* or *b-cipher* if the cost of calculating any key k_i is bounded by a limit independent of i. The simplest case is $k_i = k_0$, $i = 1, \ldots, n$. Thus, text segments may be transformed by a b-cipher in random order without any cost penalty.

(b) T is a *progressive* or *p-cipher* if the cost of calculating each key k_i from f is an increasing function of i, but successive keys may be calculated from previous keys at a cost independent of i. The keys may, for example, be calculable by a recurrence relation of the form: $k_i = g(k_{i-1})$. Such ciphers are only economical when applied sequentially to the segments of a text string.

Both types of ciphers have been used to encrypt computerized information. Examples of each follow.

(a′) A b-cipher method developed by Feistel was applied to protect information being transmitted in a remotely accessed computer system [4, 8]. The cipher is a "mixing" transformation which scrambles text segments by passing them through a series of stages which perform character transformations, bit permutations, and additions programmed by the cipher key. The cipher was programmed to transform 16 byte segments ($w = 128$) with a key of the same size. Processing was found to require 9 milliseconds per segment on an IBM 360/67.

(b′) A p-cipher based on the so-called Vernon cipher was used by Carroll to encrypt files stored in a multiprogramming system [2]. A Vernon cipher transforms a text by performing an exclusive OR (denoted $\oplus$) between it and a key string

[1] The text may have to be padded with extra bits if its length is not a multiple of w; if so, these bits should be a random pattern; a fixed one would aid a cryptanalyst.

(the concatenated segment keys). If the string is truly random and used just once, then this cipher is theoretically unbreakable. One must, however, both store and protect such key strings for later decipherment. Carroll avoided this problem by using successive outputs from a pseudorandom number generator as the segment keys, hence needed only to store the seed value used to start the encipherment of each file. When programmed for a PDP-10/50, the method required about 54 microseconds (real time) per 35-bit segment using a 36-bit cipher key.

There is essentially no data comparing the costs of b- and p-ciphers with similar work factors. One may hypothesize that, as with the examples just given, p-ciphers tend to be more quickly computed, to use shorter cipher keys, and to transform smaller size segments than do b-ciphers. In essence, a p-cipher depends for its unbreakability on continually changing segment keys, which can in many cases be algorithmically generated from a small cipher key, and be applied to individual bits or characters (e.g. exclusive OR is a bit-parallel operation). A b-cipher, on the other hand, depends entirely on the inscrutability of the transformation used, since it is repetitively applied with the same keys to successive segments; and this is best done by performing a complicated series of mixing operations to a large segment of text bits at once.

3. THE PROBLEM OF KNOWN PLAIN TEXT

A promising step in attacking any enciphered file is to establish that certain plain text segments have been encrypted. If the segments are large enough and the corresponding cipher text can be located, then many ciphers can be readily broken [7, 9].

The existence of particular plain text may be established in any of several ways. The problem of known formatting information has already been mentioned, as has the planted record problem. Other possibilities are to eavesdrop on the activities of legitimate file users, or to exercise priviledges to inspect portions of the file (allowed, for example, by privacy laws governing personal data banks).

An intruder will sometimes be seeking to extract specific information from the file, say, that relates to a particular individual or product. To reduce the danger from known plain text, a large file may be split into a number of subfiles which are individually encrypted using different cipher keys. The desired information will not be obtainable unless the known information happens to be in the correct subfile.

However, it may be possible in some cases to obtain plain text which must lie in the same subfile as the sought information. In a medical information system, for example, where complex records are maintained for each patient in a health care system, various users—doctors, nurses, hospital administrators, medical researchers, etc.—will have varying rights to access and modify records. Thus an administrator, who would have no right to medical data, could use his authority to inspect billing information for a particular patient. This information may be enough to construct sufficient plain text to break the encipherment of this patient's record.

It will have to be assumed in most applications that some plain text will eventually become known to an intruder. His next problem is to locate the corresponding

cipher text, which may be quite costly if a large file must be scrutinized. This is made much easier if the known information is associated with an update to the file.

Consider the case of the insertion of known plain text into a sequential file when the cipher text before and after can be obtained. If the cipher key is unchanged, then locating the cipher text corresponding to this plain text is readily done by comparing the cipher text versions until a discrepancy is found.

If the cipher happens to be a Vernon-type p-cipher, then all plain text and key segments following the alteration can be trivially calculated, even if a truly random key string is used. To see how this is done, let $Q = q_1 \ldots q_n$ be the n plain text segments in an index, to which a new element containing m segments is added after segment l, resulting in an updated plain text $Q' = q_1' \ldots q_{n+m}'$. Then we know that

$$q_i = q_i', \quad \text{for } i = 1, \ldots, l,$$

$$q_i = q_{i+m}', \quad \text{for } i = l+1, \ldots, n, \tag{1}$$

and $q_{l+1}' \ldots q_{l+m}'$ are the known plain text segments.

Let $K = k_1, k_2, \ldots$ denote the segment key sequence used to encrypt both Q and Q'. The cipher texts are:

for Q: $C = c_1 \ldots c_n$, where $c_i = q_i \oplus k_i$, $i = 1, \ldots, n$, and

for Q': $C' = c_1' \ldots c_{n+m}'$, where $c_i' = q_i' \oplus k_i$, $i = 1, \ldots, n+m$.

We may immediately calculate using the known plain text:

$$k_{l+i} = q_{l+i}' \oplus c_{l+i}', \quad \text{for } i = 1, \ldots, m. \tag{2}$$

But these keys were used to encrypt Q also; hence we have:

$$q_{l+i} = k_{l+i} \oplus c_{l+i}, \quad \text{for } i = 1, \ldots, m. \tag{3}$$

Equation (1) implies that we have the next m plain text segments in Q': $q_{l+m+i}' = q_{l+i}$ for $i = 1, \ldots, m$. Hence the decipherment of eqs. (2) and (3) can be repeated using $l + m$ in place of l. In this way, all text segments after position l can be deciphered.

The situation can be even bleaker when the method for generating key segments can be run backward. Carroll's random number method, for instance, computed successive segment keys by the relation $k_{i+1} = (k_i + k_{i-L}) \bmod (2^W)$, where $16 \leq L \leq 79$.

If L is smaller than $n + m - 1$, then the value of L can be determined by examining the segment keys obtained above. One may then work backward to k_1 using the relationship $k_{i-L} = (k_{i+1} - k_i) \bmod (2^w)$.

The preceeding example illustrates a fundamental principle of cryptological practice: even very formidable looking ciphers may be susceptible to simple attacks when not applied carefully. (Here, the fault stemmed from the reuse of the cipher key.)

Countermeasures to cipher breaking after updates are clearly required. One way is to change the cipher key as often as possible—after each update when feasible. A second is to perform a number of updates simultaneously to obscure the location of known plain text segments. The update problem is reconsidered in Section 6.

4. B-TREES

B-trees are a form of multiway tree suitable for maintaining ordered indexes in pseudorandom access secondary storage. The search, insertion, and deletion algorithms are designed to minimize the required number of accesses to external storage. In particular, if the number of elements in the index is N, and if k is a natural number (discussed below) determined by the characteristics of the storage device used, then each operation takes a time which is proportional to $\log_k N$. Detailed information on B-trees can be found in [1], [6], and [10].

A directed tree $\mathfrak{I}$ is a B-tree in the class $\tau(k, h)$ if $\mathfrak{I}$ either is empty ($h = 0$) or has the following properties:

(i) Each path from the root to a leaf has the same length h, called the *height* of the tree.

(ii) Each node contains n index elements, where $k \leq n \leq 2k$, except for the root, where $1 \leq n \leq 2k$.

(iii) Each node with n index elements has $n + 1$ sons, except leaf nodes, which have none.

The nodes are stored individually on pages in secondary store. Within each page P the index elements are arranged in increasing order by name: $x_1, x_2, \ldots, x_n$. Furthermore, P contains $n + 1$ pointers to the sons of the node: $p_0, p_1, \ldots, p_n$, which are undefined when P is a leaf. A page is then logically arranged as in Figure 1.

The relationship between pointers and names is constrained as follows. If p_i is any pointer on page P, then $X(p_i)$ is defined as either the empty set if P is a leaf, or as the set of all index names in the maximal subtree of $\mathfrak{I}$, the root of which is the node to which p_i points. We then require that for each page P with n elements:

$$(\forall y \in X(p_{i-1}))(\forall z \in X(p_i))\ (y < x_i < z), \quad \text{for } i = 1, \ldots, n. \tag{4}$$

An example of a B-tree is shown in Figure 2. The rectangular boxes represent pages, and the numbers to their upper left are page numbers for secondary storage. The element names are the values 1 through 25 appearing within the boxes, and defined pointers are explicitly represented by arrows between pages. The associated information fields are omitted.

In the following discussion, x is the name of an element being sought, inserted, or deleted, and the reader is assumed to be familiar with basic concepts about binary search trees.

Retrieval may be performed using property (4) in a manner analogous to binary search tree retrieval. Beginning with the root page, search the page looking for x and halt successfully when found, or unsuccessfully when not found and this page is a leaf. Otherwise, using (4) to determine in which subtree x must be located, take the root of that subtree as the next page to be examined.

One should note that retrieval of all elements in a B-tree in name order is possible through the obvious generalization of postorder traversal of binary search trees.

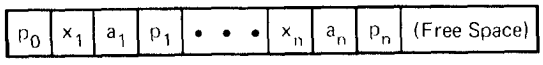

Fig. 1. Structure of a B-tree page

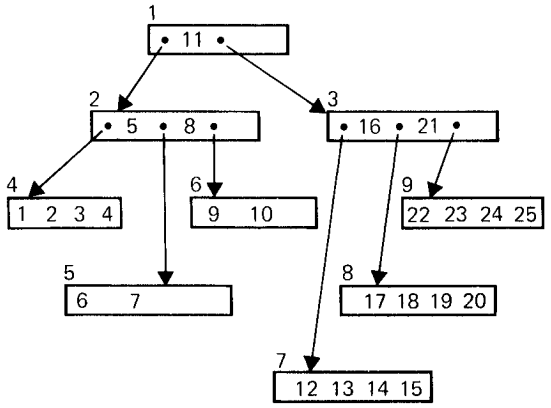

Fig. 2. Sample *B*-tree

Two nodes with the same father are called *adjacent brothers* if in the father their pointers are adjacent. Both insertion and deletion may cause a redistribution of elements in adjacent brothers. In Figure 2, pages 4 and 5 are adjacent brothers. The elements could be redistributed so that (1, 2, 3) appears in page 4, (5, 6, 7) in page 5, and 4 in page 2 to separate the names in the two sons. Note that after a redistribution, both brothers and the father should be rewritten to secondary store.

An insertion is always made at a leaf which (except when 3 is empty) is chosen to satisfy (4) as the leaf on which the retrieval algorithm fails to find x. When the page into which an element is to be inserted is already full ($n = 2k$), then space is found either by a redistribution when an adjacent brother is not full, or by splitting. Splitting involves creating a new adjacent right brother, and moving the last k elements into this node. The middle index element (x_{k+1}, a_{k+1}) and a pointer to the new brother are then inserted in the father of the page which was full. If full pages are also encountered during such secondary insertions into fathers, and no redistributions are possible, then splitting will propagate back to the root, creating h new brothers as well as a new root (hence increasing h by one).

Deletions always begin on leaves. If x is found on a nonleaf, then the leaf containing the lexical successor to x is located (in a way analogous to finding postorder successors) and these two elements are interchanged. Now x may be deleted from the leaf, and property (4) is preserved. When a leaf (not also the root) becomes less than half full, the *B*-tree structure may be recovered either by redistributing elements with an adjacent brother or by catenation. Catenation involves the merging of elements in two adjacent brothers and the separating element in their father into one brother, and the deletion of the other page from the tree. The secondary removal of an element from the father may also cause it to have too few elements. If half-full pages are encountered during each such secondary deletion and no redistributions are possible, then $h - 1$ adjacent brothers will be deleted along the path from the leaf to the root. (The root will be deleted if it had only two sons, which were catenated into one along with its single element.)

We have just described the simplest form of *B*-trees. For real applications a format without pointers should be used for leaf pages. This causes no significant

changes in the above algorithms. Another useful variation is the B^*-tree described in [10].

5. ENCIPHERMENT SCHEMES FOR PAGED FILE STRUCTURES

Paged file structures are ones which are stored as sets of fixed size pages in secondary storage. They include B-trees and its variants as special cases. The following objectives seem appropriate when designing schemes for encrypting such files, whether they are stored in the form of B-trees or in other forms:

(i) The encipherment method should not materially alter the basic algorithms for retrieval, traversal, insertion, and deletion.

(ii) The method should be applicable whether a p-cipher or a b-cipher is to be used.

(iii) The method should also be suitable when the transformation can be performed by hardware in the data channel as pages are transmitted between main and secondary storage.

A cipher may be globally applied to a paged file structure by treating the contents of each page as a plain text string to be enciphered using the identical key. This is, however, quite undesirable, because the uniform encipherment of all pages is likely to provide many cipher breaking clues.

We propose, instead, the following "page key" scheme for encrypting paged file structures such as B-trees. Let the m pages of a file $\mathfrak{I}$ have page numbers (used to locate the pages on secondary store) p_i, $1 \leq i \leq m$, and denote the plain and cipher text versions of the ith page by, respectively, $Q(p_i)$ and $C(p_i)$, $1 \leq i \leq m$. Each page p_i has also a *page id* $\hat{p}_i$.

For file encrypting we select two ciphers, a *text cipher* U which must be reversible and may be either a p- or a b-cipher, and a *page key cipher* E which need not be reversible, as well as a *file key* k_E for E. Given the page id $\hat{p}_i$, for any page p_i, $1 \leq i \leq m$, E is used to calculate the corresponding *page key* k_{p_i} as $k_{p_i} = E(\hat{p}_i, k_E)$. Page contents are then encrypted via the transformation: $C(p_i) = U(Q(p_i), k_{p_i})$, and decrypted via $Q(p_i) = U^{-1}(C(p_i), k_{p_i})$.

Page numbers will, of necessity, be available during file processing, so there remains the choice of how the page ids are derived. This may be done in two principal ways.

Method C—the pure cipher method: The page ids are directly equated with the page numbers: $\hat{p}_i = p_i$, so that the page keys may be calculated as $k_{p_i} = E(p_i, k_E)$. In this case all page keys are fixed once the file key is chosen.

Method T—the page id table method: Arbitrary values are chosen for the page ids and these are then stored in a table ordered by page number; a satisfactory method for selecting page id values will often be to read the value of a real time clock. When a page key is needed, the page id will have to be retrieved from this table as a preliminary step. On the other hand, an advantage is that individual page keys may be altered by modifying the page id table.

Note that with either method the page keys need never be explicitly stored, but can be readily derived given the file key. Furthermore, the page id values are not sensitive data which must be concealed from intruders; only their enciphered form, which is determined by k_E using E, is sensitive. But k_E is hard to deduce precisely because the page keys are never stored.

The use of different cipher keys for different file pages enhances cipher security. Some cryptanalysis schemes require relatively long cipher text strings, which can never occur because of the fixed size of pages. Also, the breaking of the encipherment of one page will give little help for the breaking of other pages, since the page keys have no direct relationship to one another.

Prior to contrasting these two page id methods further, we should determine whether or not the page key scheme fulfills the general objectives for encrypting paged files, and particularly *B*-trees.

Generally, transformations cannot be performed in the data channel by currently available hardware. Thus pages will have to be enciphered before writes, and will reappear in encrypted form in main storage after reads. The discussion in Section 7 concerning operations on enciphered, sequentially ordered indexes shows that encrypted pages of *B*-trees, which are essentially sequential indexes, can be handled without great difficulty (but see Section 8); hence conditions (i) and (ii) are fulfilled.

If the transformation U can be applied by the data channel, then condition (iii) can be satisfied, since either page id method allows the computation of the appropriate page key before a channel operation is initiated. In this case, condition (i) is trivially fulfilled, whereas condition (ii) is met provided the text cipher U can be calculated at least as fast as the channel transmits data.

Method C is preferable to method T because it requires neither extra storage for page id values nor extra steps for page id retrieval. Counterbalancing this is the property that page keys cannot be individually changed, which will be seen in Section 6 to preclude a good defense against "page breaking" after updates.

Method T may fit nicely into some paging schemes which already maintain page tables and have room for page id storage. Alternatively, it may be feasible to store the page ids in the file along with the page pointers. In the case of linked, directed tree structures such as *B*-trees, the ids may be easily maintained since there is precisely one pointer to each page (except the root) which is stored in the respective father.

When discussing *B*-tree encipherment further in the following sections, we shall often call the file key k_E the *tree key*.

6. SECURITY THREATS TO ENCIPHERED *B*-TREES

Any attack on an encipherment will have to exploit some weakness in the control of access to cipher key information or to file contents, either in primary or in secondary storage. Security of key information depends on restricting knowledge of, fundamentally, the tree key and, secondarily, the page keys. Both must be secured by conventional protection mechanisms while in main storage. The page keys themselves will never appear in secondary storage, but the tree key probably will, e.g. in a system's file directory, and should, for instance, be concealed in another encrypted file.

When direct attacks on cipher keys are prevented, cipher breaking must rely on access to cipher text and, when possible, knowledge of actual or probable plain text. Thus access to file contents should be strictly as well as circumspectly controlled. A target whose defense may be overlooked is, for instance, the backup copies of a file; these provide marvellous snapshots of ciphered versions of the file.

An intruder may, after obtaining access to cipher text, be able to exploit the potential recognizability of either the leaves, which do not contain any pointers, or the root, which may be less than half full. For example, particular leaf pages in B-trees must contain the first and the last k index elements. Thus, differentiating features of nodes should be obscured in cipher text.

One of the main features of the page key idea is that even if a cryptanalyst manages to break the encipherment of one page, this information should provide little assistance in breaking further pages. There are, however, three avenues open to further cipher breaking in B-trees.

(1) If the broken page is not a leaf, then the name and pointer information may allow the intruder to guess some of the contents of son pages. This danger is especially acute if the sons are leaves, since the name ordering property (eq. (4)) completely restricts page contents. For a B-tree $\mathfrak{I} \in \tau(k, h)$ with $h \geq 3$, the probability that a random broken page is a leaf is $k/k + 1$ or better. However, given that it is not a leaf, the probability that its sons are leaves may be as high as $2k + 1/2k + 2$.

This threat arises because of the B-tree structure, and is thus not easily countered. One possibility would be to store already encrypted—say by the E cipher—page pointers in the plain text contents on pages to make the location of sons of broken pages more difficult.

(2) Given that an intruder has obtained the key for a particular page and can periodically reread the cipher text of this page, he may continue to receive new information provided the page key remains valid. This occurs, of course, because page contents are intermittently altered due to insertions and deletions. In addition, excellent clues for breaking further pages will be provided by the changes made during splitting, catenation, and redistribution. Suppose, for example, that a broken page had $2k$ elements when deciphered at one instant, and then has, due to a split, only k elements when deciphered a short time later. Except possibly for the inserted element, the entire plain text of the new page is then known. Such information will undoubtedly allow the new page, once found, to be broken and its page key deduced. The intruder may now gaze at the file through two "broken windows," thus doubling the rate at which he can acquire new information.

The specific defense against this broken window threat is to change page keys after every page modification, if possible, or at least after every "major" modification involving splitting, catenating, or redistributing. This can be done without large added costs when the page ids are maintained via the table method. With the pure cipher method, however, the page keys cannot be changed without changing the tree key, which will probably only be convenient during the course of some global tree operation such as traversal. The alternative is to move the node contents to a new page and hence a new page key, leaving the old contents in place in the old page to give the intruder no clue that a change has been made until the tree key can be conveniently changed. This stratagem may, however, result in an unacceptable number of idle pages between tree key changes.

(3) When a page p is broken, the intruder will probably be able to deduce the page key k_p and also the page id $\hat{p}$. Since $k_p = E(\hat{p}, k_E)$, he might then try to invert E to obtain the tree key. If the set K_E of possible tree keys is not too large

(e.g. the intruder may know that certain heuristics are used to choose the tree keys), he may even attempt an exhaustive search for $k' \in K_E$ such that $k_p = E(\hat{p}, k')$.

One defense is to make K_E very large. The additional costs to store file keys and calculate E would be quite small. An additional defense is to choose E as an irreversible many-to-one cipher such that for each $\hat{p}$ a number of tree key values will result in the same page key, but this set of "equivalent" tree keys will vary with $\hat{p}$. That is, given k' and $\hat{p} \neq \hat{q}$, if $k_p = E(\hat{p}, k')$, then in general $k_q \neq E(\hat{q}, k')$ unless k' is the real tree key.

7. SEQUENTIAL INDEX ENCIPHERMENT

We postponed discussion in Section 5 of how the text cipher U is to be applied to the individual pages of a B-tree, which are essentially ordered indexes. We need to consider both how the encipherment affects access and update algorithms and what overhead encipherment and decipherment impose.

The information in an ordered sequence of N index elements (Figure 3) may be secured by dividing the text into appropriate size segments and encrypting these with a p- or a b-cipher. The question then arises whether it is better to use a p- or a b-cipher, given that one of each type exists with an acceptable work factor? This may be resolved by answering two cost questions: what is the decipherment cost when an index element is retrieved by name, and what is the (re)encipherment cost when an element is inserted, deleted, or modified?

We assume that the index contains n text segments and that each element comprises an integral number of these; thus the average element size is $m = n/N$. Let the per-segment cost of encipherment or decipherment be C_p for the p-cipher and C_b for the b-cipher.

Costs with a p-cipher. The index must be sequentially deciphered from the beginning; hence a linear search should be conducted until the desired element is located. Assuming retrieval of random names, the average retrieval cost with a p-cipher will be

$$R_p = (n/2)C_p. \tag{5}$$

An update will, in general, change the length of the index, and hence cause the portion after the change to be reenciphered. The decipherment and reencipherment cost per segment will be between C_p and $2C_p$, since the key sequence need be generated only once for both operations; let this cost be $(1 + g)C_p$, $0 \leq g \leq 1$. The total update cost is given approximately by

$$U_p = n(1 + g/2)C_p. \tag{6}$$

Costs with a b-cipher. Elements may be located using a binary search, since text segments can be independently deciphered. Further, when an update is made, only the element affected need be enciphered. (Note the assumption that index elements are contained in an integral number of text segments is critical here.)

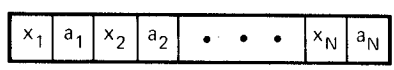

Fig. 3. Sequential ordered index

About $\log_2 N - 1$ elements on the average will have to be deciphered during a (successful) search, and one element enciphered for most updates. The retrieval cost, R_b, and the total update cost, U_b, are then approximately

$$R_b = m(\log_2 N - 1)C_b, \tag{7}$$

$$U_b = m(\log_2 N)C_b. \tag{8}$$

The retrieval and update cost estimates in (5)–(8) may now be compared to determine when one cipher type is preferable to the other, resulting in:

$$R_b < R_p \quad \text{provided} \quad C_b < ((N/2)/\log_2(N/2))C_p, \tag{9}$$

$$U_b < U_p \quad \text{provided} \quad C_b < (N/\log_2 N)(1 + g/2)C_p. \tag{10}$$

For the examples in Section 2, the cost ratio C_b/C_p was about 50, while g was probably near 0.5 for Carroll's algorithm. The preceding comparisons indicate that Feistel's b-cipher is better for retrievals only when N is about 1000 or more and for updates only when N is about 330 or more.

Table I lists for several index sizes and $g = 0.5$ the "crossover" values e_R and e_U of the cost ratio C_b/C_p, such that a b-cipher is preferable to a p-cipher for retrieval (update) when the cost ratio is smaller than e_R (e_U). If the ratio of retrieval to update operations were known, then a "mixed" crossover value could be calculated instead.

The cost estimates (5)–(8) will have to be modified to reflect special processing possibilities. For instance, when the index element fields are fixed in size, decipherment costs can be reduced, since only name fields must be examined during the search. This will reduce costs more for b-ciphers than for p-ciphers, since the key sequence for the omitted fields will still have to be calculated for the progressive ciphers, unless one chooses other storage representations for the pages.

We omitted, when deriving b-cipher cost estimates, to consider whether a binary search can be applied to a sequentially ordered file when the fields are of varying length. While a normal binary search cannot be used, it is interesting to see that such a search can, nevertheless, be carried out without the use of auxiliary tables to locate fields.

Table I. Crossover Points for Cost Ratio C_p/C_b ($g = 0.5$)

Index size N	Crossover values	
	e_R	e_U
16	2.7	6
32	4	9.6
64	6.4	16
128	10.7	27.4
256	18.3	40
512	32	85.4
1024	56.9	153.6
2048	102.4	279.3

Fig. 4. Sequential index with field delimiters

The trick is to use special characters, say * and +, to delimit fields of the index as shown in Figure 4. This allows searches to occur with the normal $\log_2 N - 1$ comparisons on the average.

Let l and r be the left and right limits (text segment numbers) of the search, $1 \leq l \leq r \leq n$, and initially $l = 1$ and $r = n$. To find a name field for a comparison step, we locate the * and + symbols which delimit it as follows. First calculate $t = (l + r)/2$, then decipher and examine the tth text segment in the index. (We assume that each segment contains at least one character.) If no * is found here, then decipher and examine successive segments to the left of the tth segment until one is. This symbol fixes the left end of the name field. The right end of the name field can be immediately fixed if a + was encountered while searching for the *. If no + was seen, then decipher and inspect successive segments to the right of the tth segment until one is.

The value of l or r will be adjusted after an unsuccessful comparison as usual. To change r to the left side of the current element is simple, since the * found delimits it. To change l to the right side, however, will require further searching to the right until the next * is located.

With this search method for variable length fields and b-ciphers, not all of each element will have to be deciphered as a rule. In order to skip over uninteresting fields using a linear search and a p-cipher, the index must be stored with length indicators preceding the fields instead of delimiter characters.

8. PAGING AND PARTIAL DECIPHERMENT

Experiments have shown that a demand paging scheme for B-trees which tries to maintain a number of the recently accessed pages in main storage buffers can be quite effective in reducing the load on backing storage [1]. If the data channel can perform encipherment and decipherment, then it is only necessary that the paging system be given the page key for each page which has been modified, so that it may be correctly reenciphered when returned to secondary storage. (When page reads are initiated, the page keys would be provided by the accessing routines requesting pages.)

When the cipher is applied by software, complications arise because pages will not normally be completely deciphered (cf. Section 7). Not only will the paging system have to know how to reencipher those pages which must be rewritten, but the details of what parts of each page have been deciphered must be available when an "old page" is given to a second or later user.

This may be done by maintaining a *decipherment descriptor* for each page retained in a main storage buffer. As shown in Figure 5, this would contain the page key k_p and a modification flag m. In addition, for a b-cipher there would be a bit string, deciph, specifying which text segments have been deciphered, and for a p-cipher there would be both an index l specifying the first enciphered segment and a key value, $k_p{}^l$, giving the corresponding cipher key segment to be used. (Note that we

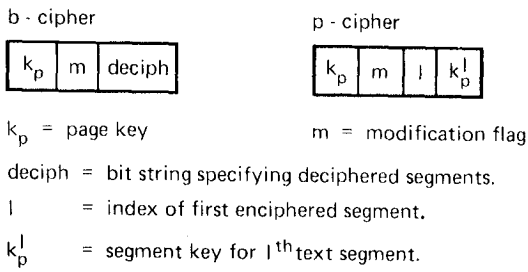

Fig. 5. Decipherment descriptors

have assumed for the *p*-cipher case that fields are deciphered serially with none skipped over.)

One should notice that the interpretation of this modification flag is not the conventional one associated with paging systems. There it is common to have the hardware set an "altered" flag whenever a page location is written into. While a page is written into during decipherment, the real information is not altered and this should not be considered a page modification by the hardware.

The decipherment descriptors could be maintained by or made directly available to the paging system. But this is undesirable as it leads to an unnecessary complication of the paging system as well as introducing yet another component whose integrity will have to be enforced. We might alternatively require that the system invoke a *B*-tree routine which would encrypt any page which the paging system wishes to write out.

In fact, it would seem that some analogous requirement would be necessary in any paging system which handles both normal and encrypted files. Since pages from an encrypted file must never appear in secondary storage in plain text form, the paging system must be able to distinguish between normal and encrypted files. In the latter case, before a write may be performed, the paging component must permit the file user to determine that the page may be safely written. This might be done by associating with each encrypted file at the time when it is opened, a procedure which may be invoked to seal pages for storage on backing store devices.

9. SECONDARY INDEXES AND DOCUMENT FILES

The design of large indexes may be complicated by either of two contingencies. There may be the need to locate associated information by secondary as well as primary index names. Alternatively, the associated information fields may be quite large compared to the name fields (e.g. complex personnel records), so that it is better to store the associated information in a separate *document file*.

The standard solution to secondary name retrieval is to create a *secondary index* which gives, as the associated information for each secondary name, the primary name in the main index. Retrieval then requires searching both indexes in succession. Encipherment, however, is straightforward using the page key method proposed, and the same or different file keys may be used for each.

When a document file is necessary, the more critical data to protect will be in that file. An effective encipherment scheme is provided by storing in each associated information field in the (main) index, both a pointer to the corresponding records

in the document file and a *document id* used to secure these records. This scheme adds a new element of flexibility since these document ids may be individually altered whenever desired.

In analogy to page ids, we should probably use these document ids not as cipher keys themselves, but to calculate the corresponding *document keys*. We may then wish to introduce a *document text cipher* in place of the file text cipher U. We shall not pursue these ideas further except to note one similarity, namely, both page ids when stored in a B-tree (Section 5) and these document ids are paired with a pointer value. Such pairs are reminiscent of capabilities [3], which typically contain both a reference to an object and a value representing rights to manipulate the object. Here, this value is the page id or document id representing the manipulation right "encipher and decipher."

10. CONCLUSIONS

We have seen that secure encipherment schemes can be developed for indexed, random access files based on the page key method. In particular, given that careful attention is paid to countering cipher breaking threats arising from changes to content and structure, enciphered B-trees may be securely handled without major changes to maintenance algorithms.

The page key method should be applicable in most systems, especially those regularly handling files through paging mechanisms. And, in fact, it may also be used with sequential files provided that these can be broken into short subfiles, and that subfile ids, analogous to page ids, can be associated with each piece.

Encipherment overhead may be a significant problem with random access files. In a sequential file, encipherment and decipherment costs can be prorated over a large number of transactions by "batch processing" them in a single pass through the file. Transactions on random access files, on the other hand, must normally be performed individually, hence incurring unproratable cipher costs. The page key method may, however, hold out the prospect of using cheaper ciphers than are feasible with traditional methods. Some simple ciphers, for example, must be rejected for use with sequential files because the normally quite long encrypted text would offer too many cipher breaking clues. The page key method uses different cipher keys for each file page, and these will not be too long, say 10^3–10^4 bytes. Thus some of the simpler and more cheaply computable ciphers may become suitable.

Encipherment costs and changes to file algorithms may be trivial if the privacy transformation can be applied during data transmission. Transmission rates for devices suitable for storing large ordered indexes may often be in the neighborhood of a million bytes per second. Even if the cipher operates on several byte text segments, it will have to be computable at a minimum rate of 10^5 segments per second. A prerequisite, then, to the introduction of data channel encipherment is the identification of good ciphers which can be computed this rapidly, along with the development of smart data channels or peripheral processors with the necessary computational power.

Until such time, the transformation will have to be applied by software. Despite the savings from partial decipherment, it is certain that the need to decipher text

during index searches will increase processing costs severalfold. It is thus necessary to seek ciphers which offer adequate work factors at acceptable costs.

There is little experimental data thus far available on the costs of programmed ciphers, and that available [5] pertains to sequential file processing. It would be desirable, therefore, to implement a *B*-tree file system which may be used as a testbed for the analysis of programmed ciphers. If the file system is properly instrumented and can be used to maintain some realistic indexes, then one may also be able to obtain some experience concerning how well the proposed countermeasures to cipher breaking threats perform.

Rather detailed versions of the proposed algorithms will be published in a forthcoming report.

REFERENCES

1. Bayer, R., McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (1972), 173–189.
2. Carroll, J.M., and McLelland, P.M. Fast "infinite key" privacy transformations for resource sharing systems. Proc. AFIPS 1970 FJCC, Vol. 37, AFIPS Press, Montvale, N.J., pp. 223–230.
3. Fabry, R.S. Capability-based addressing. *Comm. ACM 17*, 7 (July 1974), 403–411.
4. Feistel, H. Cryptography and computer privacy. *Sci. Amer. 228*, 5 (May, 1973), 15–23.
5. Friedman, T.D., and Hoffman, L.J. Execution time requirements for encipherment programs. *Comm. ACM 17*, 8 (Aug. 1974), 445–449.
6. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1972.
7. Mellen, G.E. Cryptology, computers, and common sense. *Proc. AFIPS* 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 569–579.
8. Smith, J.L., Notz, W.A., and Osseck, P.R. An experimental application of cryptology to a remotely accessed data system. Proc. ACM 27th Nat. Conf., 1972, pp. 282–297.
9. Turn, R. Privacy transformations for databank systems. *Proc. AFIPS* 1973 NCC, Vol. 42, AFIPS Press, Montvale, N.J., pp. 589–601.
10. Wedekind, H. On the relation of access paths in a data base system. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 385–397.
11. Wilkes, M.V. *Time-Sharing Computer Systems.* American Elsevier, London, 1971.