Abstract to appear in Proceedings of the 27th ACM Symposium on the Theory of Computing, May 1995.

Incremental Cryptography and Application to Virus Protection

Mihir Bellare* Oded Goldreich† Shafi Goldwasser‡

Abstract

The goal of *incremental cryptography* is to design cryptographic algorithms with the property that having applied the algorithm to a document, it is possible to quickly update the result of the algorithm for a modified document, rather than having to re-compute it from scratch. In settings where cryptographic algorithms such as encryption or signatures are frequently applied to changing documents, dramatic efficiency improvements can be achieved. One such setting is the use of authentication tags for virus protection.

We consider documents that can be modified by powerful (and realistic) document modification operations such as insertion and deletion of character-strings (or equivalently cut and paste of text). We provide efficient incremental signature and message authentication schemes supporting the above document modification operations. They meet a strong notion of tamper-proof security which is appropriate for the virus protection setting. We initiate a study of incremental encryption, providing definitions as well as solutions. Finally, we raise the novel issue of "privacy" of incremental authentication schemes.

1 Introduction

Basic cryptographic primitives such as encryption and signatures (private or public key) have received thorough theoretical treatment. In various works strong definitions of security have been proposed and achieved under general complexity assumptions. The main problem that remains and which to a large extent prevents more widespread use of strong cryptography is the inefficiency of existing schemes.

Incrementality is a new *measure* of efficiency which is relevant in a large number of different settings. We provide a comprehensive treatment of incremental cryptography. We begin by identifying and stressing conceptual issues, providing definitions for the security and efficiency of incremental primitives. We follow this up by presenting concrete, secure schemes for various tasks which (in several cases) are efficient enough to be practical, and demonstrate this by attention to issues like appropriate instantiation of abstract primitives and exact security analyses.

1.1 The Setting

A document undergoing a cryptographic transformation often does not exist in isolation: the document D that is being transformed (eg. signed or encrypted) is a modification of previous versions of the same document which have already undergone the same cryptographic transformation, or is constructed out of several other already transformed documents in some simple way. Moreover, the amount of modification that the document undergoes is often small in comparison with the total size of the document.

Examples of such settings abound; here are some. The sending of documents which are slight variations of one another to different recipients, such as a standard contract or offer being sent by a corporation; exchanges between different parties of drafts of the same document where each draft is only slightly different from the previous one; remote editing of texts or programs which must be authenticated at every change; video transmission of images which have not changed much between frames.

A particularly good example is the use of authentication tags for virus protection. Consider a tamper-proof processor with limited amount of *secure local memory*. It accesses files stored on a (possibly insecure) *remote medium* (e.g. a host machine or a WWW server). A virus may attack the remote host, and inspect and alter the contents of the remote medium (but it does not have access to the processor's protected local memory). To protect his files against such viruses, the processor computes for each file an authentication tag, depending on a key which is kept in the (safe) local memory. A virus tampering with the file can't re-compute the tag, and verification of the tag will thus detect tampering. Now note that for this to work, the processor must re-authenticate his files when he modifies them. Clearly, it is desirable to be able to update the authentication tag rather than always having to re-compute it from scratch. This problem is especially complex when the local memory is not large enough to hold (even temporarily) a single file or when it is too expensive to bring in the entire file. We note that files in this virus setting can grow very large and be subject to frequent updates; eg. consider a database being periodically altered.

The idea of incremental cryptography, as we outlined in [BGG], is to take advantage of such settings, and find ways to compute the cryptographic transformation on a document D not from scratch, but rather, somehow, as a fast function of the values of the cryptographic transformation on the documents from which D was constructed. When the "changes" are small, the incremental method may be anticipated to yield considerable advantages in efficiency.

* IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA. e-mail: mihir@watson.ibm.com.

† Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. e-mail: oded@wisdom.weizmann.ac.il. Partially supported by grant No. 92-00226 from the US–Israel Binational Science Foundation (BSF), Jerusalem, Israel.

‡ Laboratory of Computer Science, MIT and Department of Applied Mathematics and Computer Science. e-mail: shafi@theory.lcs.mit.edu .

1.2 The Issues

One quickly sees that incremental cryptography is a vast subject. Let us outline some important issues and our contributions in this regard.

PRIMITIVES. One can consider incrementality for any cryptographic primitive. The ones we focus on are signatures (private and public key) and encryption (private and public key). We focus on incrementality for the transformations themselves, namely the signing or encrypting, but discuss also incrementality of the "conjugate" transformations, namely verifying and decrypting.

TEXT MODIFICATION OPERATORS. We discuss the modification to a document in terms of applications of a fixed set of underlying document modification operators. For example: replace a block in the document by another; insert a new block; delete an old one. We then focus on the design of incremental algorithms for each such operator.

Operators should be powerful enough to reflect realistic document changes: replace, insert and delete taken together are a good choice from this point of view. In settings such as text editing, one often pulls text from one document into another. Accordingly we also consider cut and paste operations, the first cutting a single document into two, and the second pasting two documents into one. We stress that the more powerful the text modification operators are the more challenging it is to design fast incremental algorithms with respect to them. See discussion of *speed* below.

INCREMENTAL ALGORITHMS. Fix an underlying cryptographic transformation T (eg. signing under some key). To each elementary text modification operation (eg. insert) there will correspond an incremental algorithm. This algorithm takes an existing document or documents; the values of T on them; a description of the modification (here a block d to insert and an index into the document for where to insert it); and possibly underlying keys or other inputs. It must compute the value of T on the resulting document. We are interested in designing schemes possessing efficient incremental algorithms.

HISTORY-FREE IS BETTER. A trivial way of achieving incrementality should at once spring to mind. Take signatures as an example, although similar issues arise with encryption. Say I have the signature $\sigma_{\rm old}$ of $D_{\rm old}$ and modify $D_{\rm old}$ by inserting a block. I can update the signature by signing the string consisting of $\sigma_{\rm old}$ and the description of the modification. This is a *history dependent* scheme. There may be settings in which this is acceptable, but typically it is not desirable. It gets prohibitive as one makes lots of changes: verification cost is proportional to the number of changes. It requires parties to store state, and signature sizes grow with time. History free schemes are better. Our definitions mandate history freeness and our schemes are all history free.

SPEED. The basic goal, of course, is that the incremental algorithm should run faster than re-computing the transformation from scratch; the faster the better. The requirement of our definition is that the incremental algorithm run in time proportional only to the "number of modification" and not proportional to the document length. For example, if s denotes the block size then updating the cryptographic form for one modification should take poly(s)-time regardless of the number of blocks in the modified document.[1] (In addition as discussed above the algorithm should be history free.) This seems an elegant requirement, and schemes achieving it are referred to as ideal. [2] For efficiency, of course, the polynomial in s should be small, and incrementality should not be at the cost of too much increase in the time to transform from scratch.

We stress that the efficiency condition becomes more challenging as more powerful modification operations are being considered. For example, the cut and paste operators allow to omit a large chunk of text from the middle of the file by three modifications, and a fast incremental signing algorithm should be able to update the signature to fit this different-looking file within poly(s)-time and independently of the size of the file and the length of the omitted chunk of text.

SECURITY. Probably the single most important conceptual issue is security. Offhand, one might say there is no new issue in security, because we are considering existing primitives (signatures, encryption) and security has already been satisfactorily defined for them. But incrementality brings in new concerns and gives rise to new definitions.

Consider the case of digital signatures/authentication schemes. It is reasonable to assume that the adversary not only has available to it previous signed versions of documents but is also able to issue text modification commands to existing documents and obtain incremental signatures of the modified ones. Such a chosen-message-attack on the incremental signing algorithm may lead to breaking a signature system that cannot be broken by restricted attacks which don't use the incremental algorithm. Furthermore, in some scenarios such as virus attacks it is in addition prudent to assume that the adversary may be able to tamper with the contents of existing documents and signatures/authentication-tags. Accordingly we will consider two notions of security; a basic one, and a stronger notion of tamper proof security which is relevant in the virus protection setting.

Consider the case of encryption schemes. The usage of incremental encryption algorithms may leak information that is kept secret when using a traditional encryption scheme. For example, take an encryption scheme which breaks the message into blocks and encrypts each block using a secure probabilistic encryption. An incremental encryption, with respect to (single block) replacement, may operate by merely encrypting the new contents of the block and keeping all other (block) encryptions unchanged; but this enables an adversary to tell which blocks have has been changed. The encryption of a message together with the encryption of the slightly modified message, leaks knowledge. It seems that we cannot hope for *efficient* incremental encryption algorithms which hide the *amount* of difference between the two documents. Yet, it is possible to have *efficient* incremental encryption which hides everything besides.

THE PRIVACY OF INCREMENTAL SCHEMES. Here is a novel issue raised in the incremental setting: the "privacy" of different versions of a document. Suppose μ is a signature of

[1] There are technical issues about how the input is accessed, but using a RAM model these things work out.

[2] Other kinds of complexities may certainly be interesting: for example, if the from scratch transformation takes $O(n^2s)$ time to compute (where n is the number of blocks in the message) then an incremental algorithm achieving $O(ns)$ would be very nice. For simplicity however we stick to our definition of ideality.

document M and μ' is a signature of slightly modified document M'. Then, it may be desirable for μ' to yields as little information as possible about the original M. What is tolerable for one application may be different than another. For example, it may be acceptable for μ' to yield the fact that M and M' differ in a single block, as long as the identity of the block is kept secret.

1.3 The schemes

We present two signature schemes and one encryption scheme.

XOR SCHEMES. We present a particularly fast message authentication (ie. private key signature) scheme based on a pseudorandom function (PRF) [GGM]. It has incremental algorithms for (single block) insert and delete which require only four applications of the underlying PRF. It achieves basic security, and it also achieves privacy. It builds on some schemes of [BGR] which were incremental for replacement.

TREE SCHEMES. The second signature scheme provides tamper proof security, and hence *is applicable to virus protection.* It works in both the private and the public key cases— a regular (ie. not incremental) message authentication (resp. digital signature) scheme is transformed into an incremental message authentication (resp. digital signature) scheme. The scheme supports not only insert and delete but also cut and paste. The updates require a logarithmic (in the message length) number of applications of the given (non incremental) scheme. It uses 2–3 trees.

ENCRYPTION. We extend ideas on software protection [Go, Os] to provide the first incremental encryption scheme. The efficiency here is however not as good as for our signature schemes.

1.4 From theory to practice

INSTANTIATION. The schemes specified above are defined in terms of "abstract" primitives: the XOR scheme can use any PRF and the tree scheme can use any standard signature scheme. Key to achieving practicality is appropriate *instantiation* of these abstract primitives by concrete ones. In particular, suitable "pseudorandom in practice" functions can be derived from DES, MD5 [Ri] and other such primitives as described for example in [BR]. The resulting XOR schemes can run within 1.05 times the speed of the most popular message authentication scheme in practice, namely the CBC MAC, with the added advantage of incrementality that the CBC MAC does not possess. Similarly, the private key version of the tree scheme can be instantiated with the CBC MAC itself as underlying message authentication code and runs commensurately fast.

Instantiation is an important issue which may merit more discussion. As indicated above, schemes (ours in particular) are typically designed in terms of abstract primitives like one-way functions or pseudorandom functions. While many abstract primitives are equivalent in theory, the right choice, when one is interested in a final practical outcome, is a primitive which combines convenience of use with the property of possessing an efficient instantiation. In particular, finite pseudorandom functions (ie. PRFs on fixed input and output lengths) are good starting points because they can be efficiently instantiated with block ciphers like DES or via hash functions like MD5 [BR]. This is the reason we discuss our XOR scheme *directly* in terms of PRFs, and our tree scheme directly in terms of MACs, rather than say they are "based on a one-way function." (The latter is true, but not useful from the point of view of efficient instantiation.)

The central dividing line in efficient instantiation is between number theoretic or algebraic functions (factoring, discrete log) and DES or MD5 type constructions. Instantiating PRFs (or MACs) via factoring or discrete log based constructions will result in schemes orders of magnitude less efficient than schemes using block ciphers (DES) or MD5 type hash functions. Whenever possible (ie. in a private key setting) one is better of with the second kind of instantiation. (One might suggest that number theory based schemes are more secure. But in truth the two are incomparable, and some researchers familiar with both types of objects even favor the block ciphers in this regard.)

EXACT SECURITY. The user of a scheme needs to know more than just that "no polynomial time adversary can break the scheme except with negligible advantage." He needs to know, for given security or other parameters, what kind of success is achieved by an adversary with particular resources such as time and queries. (This affects efficiency, because if, for a particular security level, the security parameter must be high, then the efficiency is less.) We take these concerns into account by providing exact security reductions: our theorems quantify the success of an adversary in breaking the underlying assumption as a function of her success in breaking the constructed scheme.

1.5 Previous work and comparison with ours

The study of incremental cryptography was initiated by the current authors in [BGG]. The primitives considered there were collision free hashing and digital (public key) signatures. The text modification operation considered was replacement (of one message block by another). Incremental schemes based on the hardness of discrete log were provided. The signature scheme met the notion of basic security. Tamper proof security was pinpointed and defined, and to provide solutions for it was left as an open problem.

We have here expanded the scope in primitives to include encryption and also the private-key versions of all primitives, which are more important in practice. We are considering more realistic and powerful text modification operations like insert, delete, cut and paste. We consider multi-document settings, not only single document ones, and extend the definitions to this case. In addition, we introduce new concerns such as history-freeness and privacy. We provide the first solutions for message authentication achieving tamper-proof security, and hence for virus protection. Our schemes can be instantiated with DES and MD5 type primitives and thus are considerably more efficient than those of [BGG] which use the discrete log.

For the problem of virus protection when the virus cannot see any authentication tags of files, the work of Karp and Rabin [KR] on fingerprints can be used.[3] Their fingerprint scheme is incremental with respect to single character replacement, but does not provide fast update for single character insert/delete.

[3]This was observed by Rabin (Ben-Or, private communication).

2 Definitions

In order to define incremental cryptographic algorithms and discuss their complexity and security, we introduce a setting which we call a *multi-document system*. The latter is defined with respect to a cryptographic scheme and a set of text-modification operations.

CRYPTOGRAPHIC SCHEMES AND THE DOCUMENTS THEY PROCESS. The documents to which cryptographic transformations will apply are sequences of blocks; formally, they are strings over an alphabet $\Sigma = \{0,1\}^\ell$ where ℓ is a parameter called the *block size*. We let $D[i]$ denote the i^{th} symbol of the document D. Our complexity estimates are in terms of elementary operations over this alphabet Σ, and translating them to bit operations requires multiplication by a $\text{poly}(\ell)$ factor. Documents may also be called *texts, messages* or *files* and we will switch irrationally back and forth in nomenclature.

Incrementality is a very general notion in that it applies to a wide variety of primitives (encryption, signatures, message authentication and fingerprinting to name a few). To avoid providing a plurality of similar definitions, we formulate below a general notion of a *cryptographic scheme* which captures all the primitives we know. Later we will define incrementality for any cryptographic scheme. Below s is the security parameter.

Definition 2.1 A cryptographic scheme is specified by a triple $\mathcal{S} = (\mathsf{Gen}, \mathsf{T}, \mathsf{C})$ of probabilistic, polynomial time algorithms.

- Algorithm Gen is called the key generator. It takes as input two parameters: 1^s and $|\Sigma|$. It outputs a pair (K', K'') of keys called the transformation key and the conjugate key, respectively.
- Both the transformation T and its conjugate C act on Σ^*, using the corresponding key as additional input. We write $\mathsf{T}_{K'}(D)$ (resp., $\mathsf{C}_{K''}(D)$) to indicate the output of algorithm T (resp., C) on input D and key K' (resp., key K''). We call $\mathsf{T}_{K'}(D)$ a cryptographic form of D. For every $D \in \Sigma^*$ and every pair of keys, (K', K''), possibly produced by $\mathsf{Gen}(1^s, |\Sigma|)$, it is the case that $\mathsf{C}_{K''}(\mathsf{T}_{K'}(D)) = D$.

We'll assume for simplicity that s and $|\Sigma|$ are recoverable from each of the keys K', K'' output by the generator on input $(1^s, |\Sigma|)$. The above definition does not address security, which is more primitive-specific.

Two kinds of generators are of particular interest. A generator Gen is called *symmetric* if $K' = K''$. This corresponds to a private key setting — the legitimate parties share a key $K' = K''$. In discussing security this key will be denied to the adversary. A generator Gen is *asymmetric* if $K' \neq K''$. This corresponds to the public key setting — one of the two keys (i.e., either K' or K'') can be made public keeping the other secret. In discussing security, the public key, but not the secret one, will be made available to the adversary.

Encryption is a canonical example, with the transformation used to encrypt and the conjugate to decrypt. For signatures or other forms of authentication, the transformation is used to signing or authenticate while the conjugate is used to verify, under the (non-standard) convention that the signature contains the document and successful verification retrieves the document rather than yield the bit 1.

TEXT MODIFICATION OPERATIONS. We consider several text modification operations. Firstly, there are the single-symbol update operations which take as input a single text and modify it according to some parameter. For example, the replacement operation, given a text $T = T[1]\cdots T[\ell]$ and parameter (i,σ) so that $i \le \ell$ and $\sigma \in \Sigma$, returns the text unchanged expect for the i^{th} symbol which is set to σ. Similarly, insertion with parameters (i,σ) inserts σ in between the i^{th} and $i+1^{\text{st}}$ position of the text, making it one symbol longer. The deletion operator, with parameter i, omits the i^{th} symbol of the text, making it one symbol shorter. More powerful update operations operate on substrings of a given text. Typical operations are delete-subtext (i.e., deleting a sequence of consecutive symbols from the text) and move-subtext (which moves a sequence of symbols from one location in the text to another). Some operations can operate on many texts at once or yield a multi-text result. For example, insertion of a subtext from one text into another.

All these operations, as well as many other natural operations, can be expressed by a constant number of *cut* and *paste* operations.[4] The latter operations take and/or produce several texts (rather than taking and producing a single text). Applying cut with argument i to a text $T = T[1]\cdots T[\ell]$ results in two texts, $T' \stackrel{\text{def}}{=} T[1]\cdots T[i]$ and $T'' \stackrel{\text{def}}{=} T[i+1]\cdots T[\ell]$. Similarly, applying paste to two texts, $T' = T'[1]\cdots T'[\ell']$ and $T'' = T''[1]\cdots T''[\ell'']$, results in the text $T = T[1]\cdots T[\ell'+\ell'']$, where $T[j] \stackrel{\text{def}}{=} T'[j]$ for $j \le \ell'$ and $T[j] \stackrel{\text{def}}{=} T''[j-\ell']$ for $j > \ell'$.

THE MULTI-DOCUMENT SETTING. In a simple setting for incremental cryptography one maintains a single document together with its cryptographic (transformed) form while the document undergoes text modifications. For applications in which the only text modifications are single symbol ones (e.g., symbol insert/delete), this setting seems adequate. In this paper we consider more powerful text modifications which may deal with more than one document at a time, for example inserting a part of one document into another document. For this purpose, we present the following multi-document setting in which several documents are maintained, along with their cryptographic (transformed) forms. A motivating application is a text editor, where the documents are files, several of them being simultaneously manipulated by operations like cut and paste.

In order to define security it will be necessary to specify documents by names— thinking of the editing setting, these are the file names.

Definition 2.2 Let $\mathcal{S} = (\mathsf{Gen}, \mathsf{T}, \mathsf{C})$ be a cryptographic scheme as in Definition 2.1, and let $\mathcal{M}$ be a set of text modification operations (e.g., $\mathcal{M} = \{\mathsf{cut}, \mathsf{paste}\}$). A document system which maintains cryptographic forms (wrt $\mathcal{S}$) under $\mathcal{M}$ is an interactive machine operating as follows.

- The system is initialized with a transformation key K', obtained by running $\mathsf{Gen}(1^s, |\Sigma|)$, where s and $|\Sigma|$ are parameters.
- In response to a create document command, with pa-

[4]For example, the move-subtext operation is easily expressed in terms of (upto six) cut and paste operations. To move l symbols from start-location $i+1$ to start-location $j+1 > i+l$ in the text $T = T[1]\cdots T[\ell]$, we cut the text at locations i, $i+l$ and j, producing the subtexts $T' = T[1]\cdots T[i]$, $T'' = T[i+1]\cdots T[i+l]$, $T''' = T[i+l+1]\cdots T[j]$ and $T'''' = T[j+1]\cdots T[\ell]$. Next, we paste together T', T''', T'' and T''''.

rameters $\alpha \in \{0,1\}^*$ and $D \in \Sigma^*$, the system creates a new document[5] with name α, sets its contents to D, and its (corresponding) cryptographic form to $\mathsf{T}_{K'}(D)$. We stress that this is done by applying algorithm T to the document D, using the key K'. In addition[6], the system associates a counter, initialized to 1, with the document α.

- In response to a document pasting command, with parameters (document names) $\alpha, \beta, \gamma \in \{0,1\}^*$, the system acts as follows
  - increments the counter of document γ;
  - replaces the contents of document γ with the text which results by pasting of the texts currently in documents α and β,
  - updates the cryptographic form of γ to fit the new contents of document γ.

  The updating of the cryptographic form of document γ is done by applying an incremental algorithm, denoted IncT, associated with the transformation T and the text modification paste. When invoked by the system, IncT is given as input the current contents of the documents α and β, the contents of their cryptographic forms, and the transformation key K'. Thus, algorithm IncT takes as input a pair of documents and a corresponding pair of cryptographic forms, as well as the transformation key (and possibly the document name and its counter value). The output of IncT satisfies

$$\mathsf{C}_{K''}(\mathsf{IncT}_{K'}((D',D''),(\mu',\mu''))) = D'D''$$

  for every D', D'', μ', μ'' so that $\mathsf{C}_{K''}(\mu') = D'$ and $\mathsf{C}_{K''}(\mu'') = D''$.

Other document modification commands are processed similarly.

The above definition has made a simplification in one regard. Thinking of an editing setting, there ought to be an explicit write operation with the semantics that the cryptographic form of a document is created (and becomes available to the adversary) only once such a command is issued. In an authentication setting, for example, this would increase the power of the adversary since unwritten documents are not considered to have been authenticated by the user and so the adversary is considered successful also in case it can authenticate these (intermediate) unwritten document. We stress, however, that our schemes maintain their security also in the more stringent setting (i.e., with explicit write/authenticate operation), but due to space limitations, we stick with the simpler model presented in Definition 2.2. The above definition does not address security which is more primitive-specific; such definitions are outlined in the two following sections.

THE COMPLEXITY OF INCREMENTAL ALGORITHMS. We are interested in the complexity of implementing document systems such as the above. We ignore the cost of implementing the text modification operations as well as the cost of copying a constant number of cryptographic forms: these are system operations of predetermined cost. Our concern is the cost of updating the cryptographic forms that is incurred in running the incremental algorithms.

First, as discussed in the Introduction, we want schemes to be history-free. We ask that the length of the cryptographic form maintained for each document is a function of the current length of the document and $(s, |\Sigma|)$, but is independent of the number of modifications the document has undergone.

Second, as ask that the running time of IncT is a fixed polynomial in the security parameter s, denoted $p(s)$, independent of the length of the document. We use a RAM, rather than TM, model of computation so that sublinear time algorithms make sense: in this model any algorithm $A(x, y, \cdots)$ has direct access to each of its inputs $x, y, \cdots$ and can address an input bit with a logarithmic size address. Furthermore, for simplicity, we assume that all documents presented to the system have length bounded by 2^s, where s is the security parameter.

Third, it is required that the complexity of effecting the conjugate transformation on the cryptographic form of a document D kept by the system, should be the same as the complexity of effecting the conjugate transformation on $\mathsf{T}_{K'}(D)$ (ie., the cryptographic form obtained by directly applying T to D).

Finally, we call a multi-document system maintaining cryptographic forms *ideal* if the incremental algorithms satisfy all of the above.

INCREMENTALITY FOR THE CONJUGATE. Our treatment captures the operation of incremental algorithms for the cryptographic transformation T (e.g., encryption or signing/authentication). A similar treatment can be provided for incremental algorithms for the conjugate transformation C (e.g., decryption or verification), except that there is no (natural) analogue to first and third concerns discussed above (i.e., history-freeness and maintaining the complexity of the conjugate operation). Specifically, an incremental algorithm for the conjugate operation is given the cryptographic form of a document D, together with a previous document D', its cryptographic form C', and a description of the modification M, by which D has been obtained from D', and returns D.

3 Incremental authentication

Here we propose incremental schemes for various forms of authentication: signatures, message authentication and fingerprinting. We will present two incomparable schemes which are both "ideal" in efficiency according to our discussions of Section 2. The first is an incremental scheme for message authentication which is very simple and (when properly instantiated) fast in practice, but is secure only in the basic sense. It supports inserts and deletes, but not cut. The second scheme is presented again for message authentication but is extendible to digital signatures as well. Although not as fast as the first scheme, it is very efficient, and achieves tamper proof security. It supports not only insert and delete but also cut and paste. It can be used for the virus protection application. We begin with the definitions of security underlying these schemes.

[5] In case a document with name α existed before, it is deleted before performing this command. We may assume, without loss of generality, that this never happens.

[6] This counter is not needed in settings in which only basic security (as defined below) is required. It seems that the counter, or some other tamper-proof register associated with each document is required in order to achieve ideal tamper-proof incremental systems.

3.1 Definitions of security

The introduction of incremental authentication raises new security issues. We distinguish two settings, or requirements, in security: basic security and the stronger notion of tamper proof security. Definitions of basic and tamper proof security, in the single-document setting, were provided in our previous work [BGG]. Here we extend both definitions to the multi-document setting.

BASIC SECURITY. Basic security addresses a setting in which a user applying the incremental algorithm is assured of the authenticity of the document and authentication tag to which the algorithm is applied— this is the natural case in which, for example, old documents and their authentication tags are stored securely on the user's machine, and the definition of [GMR] extends in the natural way. Specifically, the adversary may obtain signatures to documents of its choice by issuing corresponding create-document commands. In addition, it is natural and certainly safer to assume that the adversary can also issue document-modification commands and so obtain the effect of the incremental signing algorithm on previously formed signatures.

A basic attack on a document system which maintains cryptographic forms (wrt $\mathcal{S}$ under $\mathcal{M}$) consists of using the system as suggested in Definition 2.2. In course of such an attack, the adversary can create arbitrary documents of its choice and obtain the corresponding signatures (produced by the ordinary signing algorithm). In addition, the adversary can issue document modification commands with document-names and parameters of its choice and obtain the corresponding signatures (produced by the incremental signing algorithm). These signatures are associated with the documents which result from the corresponding modification commands. To be deemed successful, the adversary must produce a signature to a document different from all the above (i.e., the documents appearing during the attack). A definition can be easily produced following the standard paradigms.

We remark that incremental signing queries may supply the adversary with information it cannot obtain by an ordinary attack (i.e., using only ordinary signing queries). This may be the case even if the adversary only uses the incremental queries as a shortcut to making ordinary signing queries (since the two signing algorithms may produce different distributions). However, the effect of incremental signing queries is more dramatic in case the adversary may tamper with documents as in the stronger definition presented below.

TAMPER-PROOF SECURITY. In some settings it is also natural to allow the adversary to tamper with the documents and signatures stored in the system and so obtain the effect of the incremental signing algorithm on arbitrarily chosen pairs of strings, which are not necessarily valid (document,signature) pairs. Thus, the second security notion, called tamper-proof security, arises.

A tampering attack is similar to the basic attack described above, except that the adversary may alter the context of documents and cryptographic forms stored by the system. Alternatively, we may describe the adversary as issuing, in addition to the above document-creation and document-modification commands, also "tampering" commands of the form $\mathsf{alter}(\alpha, D, C)$, for $\alpha \in \{0,1\}^*$ and $D, C \in \Sigma^*$ of its choice. The effect of such a tampering command is that the system changes the contents of the document named α into D and the contents of its cryptographic form into C.

A tampering attack is natural in some settings (e.g., fingerprinting for virus protection). We stress that the incremental signing algorithm does not necessarily check that the old signature is valid before modifying it according to the required update. (Typically, the incremental signing algorithm may not have enough time to verify the validity of the old signature.)

Tampering not only provides the adversary with more power, it also raises a definitional problem. Suppose that the adversary obtains a signature by effecting the incremental signing algorithm on an invalid (document,signature) pair. The question is to which document do we associate the "signature" produced this way (which may not be a valid signature to any document). Before proceeding, we remark that this question is important since forgery is defined as ability to produce signatures to documents not encountered so far (and thus it is crucial to properly define which documents have appeared so far). Our convention, by which documents are accessed by their name, plays a major rule in resolving this question.

As discussed above, once signatures are produced for tampered documents (via the incremental signing algorithm) it is not clear to which document-contents to associate them. Our convention, justified below, is to associate these signatures to the contents which would have resided in these documents (document-names) if they were not tampered with. Namely, although the adversary may tamper with the documents and alter their contents, the signatures that it obtained are associated with the untempered documents (thus ignoring the tampering). Thus, we associate with a tampering-attack two sequences of documents. One is the actual sequence of strings appearing as contents of the various documents, in various times. The second sequence is the sequence of virtual documents defined as follows. The virtual document at the moment of issuing a document-creation command is the document specified in the command (for which a signature is obtained via the ordinary signing algorithm). The virtual document at the moment of issuing a document-modification command is the document resulting by applying the command to the virtual documents the names of which are specified in the command. We stress that virtual documents are not affected by tampering commands (although the actual documents are effected). To be deemed *successful*, the adversary must produce a signature to a document different from any *virtual document* defined by the attack.

Our justification for defining forgery with respect to the virtual documents is that the decision to sign a requested document is made by the application level which is likely to relate to these (virtual) documents rather than to the actual documents (handled by the cryptographic system level). Our choice is particularly justified in the context of fingerprinting for virus-protection – see below. We conclude this subsection by remarking that in this context (i.e., fingerprinting for virus-protection) the adversary is deemed successful if it can produce a document different from the current *virtual* document so that the fingerprint for the two documents are the same. Note that in this context, it is not required that the produced document did not appear as a virtual document in the past.

3.2 The XOR schemes

Background. Some existing message authentication schemes offer a natural incremental algorithm for the replacement operation such that basic security is ensured. [7] Supporting insertion and deletion is harder, even just for basic security, and no existing scheme of which we are aware achieves it. We introduce simple schemes supporting insertion and deletion. They extend the XOR schemes of [BGR] which were incremental for replacement. The chaining technique we introduce is quite general.

The scheme. The key, denoted a, held by the parties is a pair (a_1, a_2) of random strings which specify pseudorandom functions $f_1 = f_{a_1}$ and $f_2 = f_{a_2}$, chosen from a fixed underlying collection of PRFs [GGM]. We let rand denote the algorithm which given a string σ picks a random k bit string r, called a *randomizer*, and returns $\sigma \,.\, r$. To authenticate message $D[1] \dots D[\ell]$ begin by prefixing it with a special start symbol $D[0]$ and postfixing it with a special end symbol $D[\ell+1]$. The authentication-tag of $D = D[0] \dots D[\ell+1]$ is computed in three steps:

(1) *Randomize:* For each $i = 0, \dots, \ell+1$ let $R[i] = \mathsf{rand}(D[i])$. We call $R = R[0] \dots R[\ell+1]$ the randomized version of D.

(2) *Chain and hash:* Let $h = \oplus_{i=0}^{\ell} f_1(R[i], R[i+1])$, and call this value the hash of D.

(3) *Tag:* Let $T = f_2(h)$ be the tag of D. This is output.

Note that formally the authentication-tag consists of all the randomizers together with the final tag. Informally, however, the randomizers are thought of as part of an extended message, and only the final tag is thought of as "the tag."

It is worth noting that the final step is necessary; if we output h as the tag of D, the scheme can be broken, by xor-ing together some legitimately obtained tags.[8]

How to increment. Now we specify how increments are performed. Suppose we want to compute the tag for the document $D' = \mathsf{insert}(D, i, \sigma)$ where $0 \le i \le \ell$. Let $R' = \mathsf{rand}(\sigma)$. The randomized version of D' is taken to be $R[0] \dots R[i] \,.\, R' \,.\, R[i+1] \dots R[\ell+1]$. The hash is updated by

$$h' = h \oplus f_1(R[i], R[i+1]) \oplus f_1(R[i], R') \oplus f_1(R', R[i+1]) \,.$$

The new tag is $T' = f_2(h')$. If we want to compute the tag for the document $D' = \mathsf{delete}(D, i)$ where $1 \le i \le \ell$, then we let $R' = \mathsf{rand}(D[i-1])$. The randomized version of D' is taken to be $R[0] \dots R[i-2] \,.\, R' \,.\, R[i+1] \dots R[\ell+1]$, and the hash is updated to $h' =$

$$h \oplus f_1(R[i-1], R[i]) \oplus f_1(R[i], R[i+1]) \oplus f_1(R', R[i+1]) \oplus d \,,$$

where $d = f_1(R[i-2], R[i-1]) \oplus f_1(R[i-2], R')$ if $i \ge 2$ and 0 if $i = 1$. Again the tag is just $T' = f_2(h')$.

Notice that incrementing requires four/six computations of a PRF together with some XORs and other simple operations. It is this considerably faster than re-computing the tag from scratch.

Security. The adversary is allowed document create operations and insert or delete operations. A corollary of Theorem 3.1 below is that if f_1, f_2 are chosen uniformly from a family of pseudorandom functions, then the document system presented above is secure in the basic sense.

For applications it is important to have more information on how the strength of the underlying PRFs translates into the strength of the scheme. Thus, the theorem itself specifies our ability to break the underlying PRFs as a function of the adversary's success probability. The proof is omitted from this abstract.

Theorem 3.1 Let f_1, f_2 have outputs of n bits and be chosen from a PRF family. Suppose the randomizing algorithm randomizes its input by appending a k bit random string. Suppose that the document system can be broken with probability p in an attack which runs in time t, makes m_c document create requests, and m_i incremental requests (insert or delete). Let $m = m_c + m_i$ and let L be the maximum length of any document involved. Then, the underlying pseudorandom function family can be broken with probability $\frac{p}{2} - O(m^2 \cdot 2^{-n}) - O((m_c L + m_i)^2 \cdot 2^{-k})$, in time $O(t + (Lm_c + m_i)(k + s + n))$, and making $O(m_c L + m_i)$ oracle queries.

We remark that the above document system is not tamper-proof secure. For example, the adversary can first ask to create and tag a document *abcde*. Next it tampers with this document converting its contents to *cde* and ask to modify it by deleting the second symbol. It obtains a valid tag for *abce* although this document did not appear in the attack. A tamper-proof secure scheme for message authentication is presented in the next subsection.

Instantiation and efficiency. As discussed in the Introduction, the PRFs are instantiated via DES or MD5, individually or in combination. One would typically choose a fairly large block size so that the extra memory required to store the randomizers is small in comparison to the document size: say 5% of the original. Now several instantiations are possible. To discuss them let b denote the size of a block in the randomized document.

One example is to use only DES, assumed to be a PRF. For a 56 bit DES key a let the PRF f_a, taking b-bit inputs, be defined by cipher block chaining— this is still a PRF [BKR]. Now note that the number of DES computations to compute the tag in our scheme is essentially just 5% more than that for doing DES CBC of the entire message. Thus we run at essentially the same speed as the most widely used existing message authentication scheme with the added advantage of incrementality.

As an aside we note that the XOR schemes of [BGR] require at least 25% more DES operations than the CBC. The fact that we have only a 5% overhead is due to the chaining and exhibits another advantage of this idea.

Another good instantiation is via composition. Again let a be a DES key. Let the PRF f_a be defined by $f_a(x) = \mathrm{DES}_a(\mathrm{DES}_a(\mathrm{MD5}_1(x)) \oplus \mathrm{MD5}_2(x))$, where $\mathrm{MD5}_1$ (resp. $\mathrm{MD5}_2$) is the first (resp. second) half of the output of MD5— this is shown by [BR] to be a PRF assuming MD5 is a collision-free hash function and DES is a PRF. In software this may be faster than the above.

Note that the scheme has additional efficiency properties. For example, MAC computation can be parallelized because the f_1 computations can be made in parallel.

[7] For example, hashing under linear universal-2 hash functions [CW] yields an incremental fingerprinting scheme. A message authentication scheme can be derived by appropriately encrypting the fingerprint.

[8] For example, xor-ing together the hash value of D and the hash values obtained (as below) by two different modifications to D, e.g., $\mathsf{delete}(D, i)$ and $\mathsf{delete}(D, j)$ for suitably chosen i, j, yields a hash value for the document $\mathsf{delete}(\mathsf{delete}(D, i), j)$.

Finally note keeping storage to within 5% of document size is just an example— it could go lower.

3.3 The search tree schemes

In this subsection we present a document system maintaining message-authentication tags, via fast incremental algorithms for tagging and verifying, with respect to powerful text-modification operations such as cut and paste. Our construction utilizes any ordinary message-authentication scheme. Assuming that the basic scheme is secure in the ordinary sense (i.e., withstands a chosen-message-attack), we show that the incremental system is tamper-proof secure.

Let MA be an ordinary message authentication algorithm and MA_a the tagging function induce by MA with the authentication-key a (e.g., $\mathsf{MA}_a = f_a$, where f_a is taken from a family of pseudorandom functions – see [GGM]). Let VMA be the corresponding verification algorithm (e.g., $\mathsf{VMA}_a(m,t)$ may merely consist of computing $\mathsf{MA}_a(m)$ and checking whether it equals t). We stress that these primitives are not incremental ones; yet, we will build incremental schemes out of them.

The main idea in our construction of an incremental tagging algorithm is to "keep the adversary under control" by virtue of partial verification. Specifically, before modifying a part of the tag, the incremental tagging algorithm checks that this part is "locally" valid. A standard construction, namely tree authentication à la Merkle [Me1, Me2], can be used to provide the first implementation of this idea. However, this only works for replacement. To handle the more complex operations discussed above we use 2-3 trees [AHU].

THE BINARY TREE SCHEME. To help the reader understand what follows the binary tree scheme is now presented. The construction may be standard, but the proof that it achieves tamper resistance is not trivial; however we'll omit it because we present and prove correct a more general scheme below. Assume for convenience that $\ell = 2^h$ is a power of two. The (incrementable) tag of a document $D = D[1]\ldots D[\ell]$ is a balanced binary tree of MA-tags. More formally, let V_h denote the set of all strings of length at most h associated in the obvious manner with the vertices of the balanced binary tree of height h. The tree of tags can be seen as a function Tag: $V_h \to \{0,1\}^*$ which assigns a tag to each node. This function is computed bottom-up as follows:

- For each i, let $\mathsf{Tag}(w) = \mathsf{MA}_a(D[i])$, where w is the i^{th} leaf.
- For each non-leaf node w, let $\mathsf{Tag}(w) = \mathsf{MA}_a(\mathsf{Tag}(w0), \mathsf{Tag}(w1))$.

Note that $\mathsf{Tag}(\lambda)$ is the tag of the root of the tree. In order to prevent replacement of one document by another (or by an old version of the same document), we redefine the tag of the root to be $\mathsf{Tag}(\lambda) \stackrel{\text{def}}{=} \mathsf{MA}_a(\mathsf{Tag}(0), \mathsf{Tag}(1), \alpha, \mathtt{cnt})$, where α is the name of the document and cnt is the current counter value (associated with this document).

The incremental tagging algorithm works as follows. Suppose $\mathsf{Tag}(\cdot)$ is the function describing the tag of D, and that the j^{th} symbol of D is to be replaced by the symbol $\sigma \in \Sigma$. We first check that the path from the claimed current value to the root of the tree is valid. Then we perform the update. Details follow.

Let $u_0, \ldots, u_h$ be the path from the root $u_0 = \lambda$ to the j^{th} leaf, denoted u_h. Then

- check that VMA_a accepts $\mathsf{Tag}(\lambda)$ as a valid authentication tag of $(\mathsf{Tag}(0), \mathsf{Tag}(1), \alpha, \mathtt{cnt})$, where α is the name of the document and cnt is the current counter value (associated with this document).
- for $i = 1, ..., h-1$: check that VMA_a accepts $\mathsf{Tag}(u_i)$ as a valid authentication tag of $(\mathsf{Tag}(u_i0), \mathsf{Tag}(u_i1))$.
- check that VMA_a accepts $\mathsf{Tag}(u_h)$ as a valid authentication tag of $D[j]$.

If these checks succeed then update Tag as follows:

- set $\mathsf{Tag}(u_h) \leftarrow \mathsf{MA}_a(\sigma)$
- for $i = h-1, \ldots, 1$: set $\mathsf{Tag}(u_i) \leftarrow \mathsf{MA}_a(\mathsf{Tag}(u_i0), \mathsf{Tag}(u_i1))$.
- set $\mathsf{Tag}(\lambda) \leftarrow \mathsf{MA}_a(\mathsf{Tag}(0), \mathsf{Tag}(1), \alpha, \mathtt{cnt}+1)$.

We stress that the values of Tag on all other nodes (i.e., those not on the path $u_0, \ldots, u_h$) remain unchanged.

THE SEARCH TREE SCHEME. Recall that a 2-3 tree has all leaves at the same level/height (as in case of balanced binary trees) and each internal node has either 2 or 3 children (rather than 2 as in binary trees). We stress that a 2-3 tree, alike a binary tree, is an ordered tree and thus its leaves are in order. Thus, storing a single symbol in each leaf of the tree defines a string over Σ. It is well-known that such trees support insert and delete (of a single symbol/leaf) in logarithmically many operations[9], where basic operations consist of any single change in the topology of the tree (i.e., adding or omitting a vertex or an edge). It is also easy to verify that a paste operation (merging two trees so that the leaves of the resulting tree represent the concatenation of the leaves of the two trees) can also be implemented in logarithmically many operations. A simple implementation of the cut operation results in at most log-square operations (which correspond to the truncation of logarithmically many subtrees). A more careful implementation enables to repair the "damages" created by a cut operation using only logarithmically many operations. Finally, we note that in order to allow fast search (i.e., locating the i^{th} leaf, given i) it is useful to append a counter to each vertex specifying the number of leaves in the subtree rooted at it. Clearly, these counters can be updated within the stated complexity. The two types of counters in the following description should not be confused: one counter, used above, representing the number of modifications to the document (hereafter referred to as version counter), while the other counter (termed size below), represents the number of symbols in a subtext rooted at a vertex of the tag-tree.

Now, the (incrementable) tag of a document $D = D[1]\ldots D[\ell]$ is a 2-3 tree of MA-tags, hereafter referred to as a tag-tree. Each node w is associated a label which consists of a tag (authenticating the children) and a counter representing the number of leaves in the subtree rooted at w. The tag of w is formed by authenticating (using MA_a) the labels the children of w, in the natural generalization of the above. Namely, the label of an internal vertex w is a pair $(\mathsf{MA}_a(L_1, L_2, L_3), \mathtt{size})$, where L_i is the label of the

[9] To insert a leaf, add it as a child to the suitable level $h-1$ vertex. In case the resulting children-degree of this vertex is 4, split it into two vertices so that both are children of its parent. The parent may be split so too, and so on until one gets to the root. If the root needs to be split then the height of the tree is incremented. To delete a leaf, we apply an analogous procedure. Namely, if the resulting parent and its siblings have total children-degree at least 4 then we rearrange these children so that each of the resulting parent nodes has children-degree either 2 or 3. In case the total children-degree is at most 3, we merge the parent and its sibling to one vertex and turn to its parent.

i^{th} child of w (in case w has only two children, $L_3 = \lambda$) and **size** is the number of leaves in the subtree rooted at w. The tag of the root is formed as the other tags, except that the information to which MA_a is applied contains also the document name and the version-counter. Verification is done analogously to the way it was conducted in the binary-tree scheme (i.e., for each vertex we check that VMA_a accepts its tag as valid authentication of the labels of its children) except that we also check that the subtree-counters of the children sum-up to the subtree-counter of their parent.

The incremental tagging algorithm proceeds as follows. Suppose that a document, so tagged, is to be cut at location j. We first locate the j^{th} leaf (using the subtree counters contained in the nodes). This takes $O(\log \ell)$ time. Then, we perform a partial validity check analogously to the way it was conducted in the binary-tree scheme except that we also check that the subtree-counters of the children sum-up to the subtree-counter of their parent. Again, we check only the validity of the tags for vertices on the path from the leaf to the root. (Note that these vertices are the parents and ancestors of all vertices which are to undergo topological changes.) When checking the tagging of the root, we use the corresponding document-name and current version-counter. If this check succeeds then we go ahead and implement these topological changes, creating new tags for the corresponding vertices. The tagging of the root is treated taking into account its slightly different structure (i.e., MA is applied here to information containing also the document name and the incremented version-counter). Incremental tagging for the paste operation is performed analogously. In both cases, incremental verification is similar.

A sketch of the proof of the following theorem is in Appendix A. We assuming for simplicity that $s = \log_2 |\Sigma|$.

Theorem 3.2 Suppose that the document system can be broken with probability $p(s)$ in an attack which runs in time $t(s)$, making document-create operations for documents of total length $L(s)$ and at most $m(s)$ document-modification operations, each producing a document of length at most $\ell(s)$. Then, the underlying message authentication system can be broken with probability $p(s)/q(s)$ via a chosen-message-attack which runs in time $O(t(s))$ and makes at most $q(s) \stackrel{\text{def}}{=} O(L(s) + m(s) \cdot \log \ell(s))$ queries.

The adaptation to signature schemes is immediate by substituting each reference to an (ordinary) authentication scheme by referring to an (ordinary) signature scheme.

Application to Virus Protection. The setting for virus protection by authentication was discussed in the Introduction. Our tamper-proof incremental message authentication scheme yields a virus-protection system in the sense discussed there. This is done as follows. Each file is stored on the insecure media together with its tag-tree. By a suitable choice of parameters the storage overhead can be negligible with respect to the file itself. For example, we can partition the file into blocks of length s^2, where s is the length of the tags (and the key) in the basic message-authentication scheme MA. For an L-bit-long file, we get a tag-tree that has $\frac{L}{s^2}$ leaves and can be encoded as a binary string of length $O(L/s)$. For each file, the user only needs to keep $O(s)$ bits (in local secure memory); these bits are used to store the key of the authentication-scheme, the file-name and its current version-counter. Whenever the file is modified the tag-tree (residing in the insecure media) and the version-counter (kept in the secure local memory) are modified as described in the incremental scheme (above). Whenever the user wishes to verify the integrity of its file, it verifies the validity of the tag-tree in the obvious manner.

The underlying message authentication scheme can be taken to be any of the standard ones. For example, the CBC MAC, or just one of the PRFs discussed above. (Any PRF is a MAC [GGM]). Our scheme has the additional property that it is secure even in face of an adversary who can see the authentication tags and even tamper with them. In contrast, the fingerprinting method of Karp-Rabin [KR] is secure only if the adversary cannot see the fingerprint.

4 Incremental encryption

4.1 The security of incremental encryption

As discussed in the Introduction, the usage of incremental encryption algorithms may leak information that is kept secret when using a traditional encryption scheme. Below, we outline a definition for the special case of incremental encryption with respect to single symbol replacement.

Loosely speaking, we say that an incremental encryption, with respect to single symbol replacement, is *secure* if given a sequence of encryptions $E_1, ..., E_t$, produced by encrypting D_1 as E_1 and deriving each subsequence E_i by incrementing the previous E_{i-1}, it is infeasible to derive any information about the original document D_1 as well as its modifications $D_2, ..., D_t$ (except the fact that D_i is obtained by replacing a single symbol in D_{i-1}). Equivalently, consider any two sequences, $\overline{A} = (A_1, ..., A_t)$ and $\overline{B} = (B_1, ..., B_t)$, so that $A_1, B_1 \in \Sigma^\ell$ and A_i (resp., B_i) is obtained by replacing a single symbol in A_{i-1} (resp., B_{i-1}). Then, it is infeasible to distinguish the sequence of encryptions produced by the document system when handling a create-command for A_1 and the corresponding replacement-commands of $\overline{A}$ from the sequence of encryptions produced by the document system when handling a create-command for B_1 and the corresponding replacement-commands of $\overline{B}$.

4.2 Schemes for incremental encryption

If we allow incremental schemes which are efficient in the amortized sense then there exist trivial solutions. Namely, the encryption of the document can be augmented by an encryption of the description of the modification, until the number of modifications equals the length of the document, at which point one can re-encrypt the document. This might be acceptable in some settings, but a non-amortized solution is worth seeking.

Another approach to incremental encryption is to use the idea of "software protection" as defined in [Go]. (The setting consists of a processor, having only a limited amount of local memory, to store and access information on an insecure remote memory. The simulation should be oblivious in the sense that the actual access pattern does not leak information about the original/simulated access pattern. The translation from oblivious simulation of RAM to an incremental encryption scheme is quite obvious: the role of the processor is played by the user, whereas the remote memory is associated with the encryption.) A software protection scheme with polylogarithmic overhead exists [Os], but is also amortized, and using this results in an incremental encryption scheme whose efficiency is in the end not better than that of the trivial solution above.

However the ideas of the software protection schemes of [Go, Os] can be adapted to derive an incremental encryption scheme for (single symbol) insert/delete that is efficient in the strict sense (i.e., number of simulation steps per original operation) rather than in the amortized sense (as presented there). The adaptation is achieved by "pipelining." A brief description follows.

The scheme maintains secure encryptions of documents which undergo a sequence of (single symbol) replacements. (The scheme is presented in terms of private-key encryption, but can be easily converted into a public-key setting.)

In our solution we use an arbitrary (private-key) semantically-secure (probabilistic) encryption scheme $\mathbf{E}$ (the key is implicit in the notation). We assume that $\mathbf{E}$ can be used to encrypt symbols in Σ as well as pairs (i, σ), where $\sigma \in \Sigma$ and i is an integer not greater than the length of documents in our system. Using $\mathbf{E}$, we first present an obvious algorithm that maintains encrypted versions of documents which undergo symbol-replacement. The encrypted versions consist of two sequences of encryption values, denoted E_1 and E_2. The first sequence, E_1, is a block-by-block encryption of some reference document $D = D[1] \cdots D[\ell]$; whereas the second sequence, E_2, encodes the sequence of modifications, denoted $M = M[1] \cdots M[t]$, by which the current document has been obtained from D. The obvious algorithm increments the encryption of a modified document by appending the encryption of the modification to E_2. Every ℓ steps the algorithm recovers the modified document and re-encrypts it using a block-by-block encryption, thus forming a new encryption sequence E_1 (and setting E_2 to be empty). The amortized complexity of this algorithm amounts to *two* block encryptions per each modification.

An important observation is that one may 'pipeline' the expensive actions of the above algorithm. Suppose first that we are allowed to keep intermediate results in some secure location (invisible by the adversary). Then, once the length of E_2 reaches ℓ, we can start preparing the new encryption of the document, denoted D', which results from D by applying the (first ℓ) modifications in M. We perform all required computation along with the next ℓ modifications, while allowing M to grow upto a total length of 2ℓ. At this point, we have the encryption, denoted E', of the document D' (yet, indeed, now the current document is different). Replacing E_1 by E' and omitting the first ℓ modifications in M, we obtain the encrypted form of the current document. We stress that, within our (realistic) model of computation, these operations (switching files and runcating a file) can be performed in constant time. To summarize, the algorithm works in epochs, each consisting of ℓ modifications. In each epoch, the algorithm updates the encryption to match the modifications performed in the previous epoch.

In the above description, we have assumed that the user can store its intermediate results (which require $O(\ell)$ space) in a location invisible by the adversary. This assumption is unrealistic in some settings and is inconsistent with our definitions as presented in Subsection 4.1. We thus turn to implement the above ideas without making this assumption[10]. To this end, we encrypt documents using three sequences of encryption values, denoted E_1, E_2 and E_3. The first two sequence, E_1 and E_2, are as above and suffice for decrypting the document. The additional sequence E_3 is an encryption of a "work area", denoted $W = W[1] \cdots W[2\ell]$, used to implement the above procedure.

Following is a description of what is being done in one epoch. (In our description, we do not mention explicitly the encryption operations, thus whenever we say that we set a symbol of W it is to be understood that the corresponding encryption is computed and stored.) First, we set W to hold the relevant information; i.e., $W[i] \leftarrow (i, D[i])$ for $i \leq \ell$ and $W[i] \leftarrow M[i-\ell]$ for $\ell + 1 \leq i \leq 2\ell$. (Here we assume that the modification records have the form (i, σ), where i is a location and σ a symbol to be placed in that location.) Next, we sort the pairs in W by their left element, hereafter referred to as their *sorting-keys*, so that if two sorting-keys are equal then the corresponding pairs are kept in order. It is crucial that the sorting is performed by an efficient and oblivious sorting network such as Batcher's sorting network [Ba]. We stress that whenever two pairs are compared and switched/unswitched they are re-encrypted by $\mathbf{E}$ (and so the adversary cannot tell if they were switched or not). This guarantees that the entire sorting procedure does not leak any information to the adversary. Once the sorting is completed, W is scanned while setting all occurences with the same sorting-key, save the last one (which is the 'newest' one), to a large dummy value (i.e., $(\ell + 1, \cdot)$). Now, we sort the pairs in W (by the sort-key) again, and obtain a sequence in which the first ℓ entries hold the updated document D'. Finally, we set E_1 to hold the encryption of D' and drop the first ℓ elements of E_2.

Using the AKS sorting network [AKS], our implementation of one epoch requires $O(\ell \log \ell)$ steps (whereas if we use Batcher's network we get a total of $O(\ell) + \ell(\log_2 \ell)$ steps). These steps can be partitioned evenly among the ℓ modification actions yielding the desired complexity.

As stated above, each of the schemes presented in [Go, Os] can be adopted to yield an incremental encryption scheme for (single symbol) insert/delete that is efficient in the strict sense. This is done analogously to the above, provided hat the document length stays within some predetermined bounds (e.g., between $\ell/2$ and 2ℓ). Namely, the encryption of a document consists of three sequences of encrypted values, E_1, E_2 and E_3, where E_1 and E_2 are as above and E_3 is an encryption of the workspace of some oblivious simulator. As above, the algorithm works in epochs consisting of ℓ modifications each. In each epoch, the incremental algorithm performs the ℓ modifications of the previous epoch. This is done by simulating a RAM which maintains a data structure enabling fast performance of insert/delete (e.g., a 2-3 tree).

5 The privacy issue

Privacy is an interesting new issue in incremental cryptography to which we provide a brief introduction here.

Security, as defined for signature and encryption schemes, is concerned with what an illegitimate/outsider party which does not know the private key can do or learn. For example, in signature schemes it was required that this outsider (called the adversary) cannot forge signatures. We now consider the information regarding *previous versions of the document* which can be inferred by the legitimate party when inspecting the current document together with the current cryptographic form. That is, suppose, for example, that we are given a document D together with its (updated) cryptographic form and we are told that D was obtained from

[10] Here is where we use the ideas of [Go, Os].

some other document, called D', by a deleting a single symbol. Perfect privacy would mean that we cannot tell the location of the deleted symbol. Partial privacy may mean that we cannot tell the identity of the deleted character (but we may have some information regarding its location).

Perfect privacy is a natural concern in the context of signatures. Suppose that one uses an incremental signature scheme to produce signatures to related commitments given to various parties. It is desirable that none of these parties can learn from the signature given to it something concerning commitments given to other parties. Partial privacy may be useful too. Suppose Alice has a standard commitment form in which she only fills-up some very few spaces before signing (many such forms do exist in the business world). Clearly, Alice should not care if Bob, to him she gave such an incremental commitment, learns that she have signed the commitment (given to him) by incrementing a signature to a different instance of this commitment, as long as Bob cannot find out any details concerning this previous commitment.

A definition of perfect privacy can be easily produced following the standard paradigms. Specifically, given a document D and a signature to it, it should be infeasible to distinguish whether the signature was by the document system in response to a create command or in response to a text modification command. Definitions of partial privacy may vary for ones in which the amount of modification is the only information being leaked to ones in which only the secrecy of replaced/deleted symbols is preserved.

Our first message authentication scheme (i.e., the XOR-scheme of Section 3.1) satisfies perfect privacy; whereas the second scheme (i.e., the tree scheme) satisfies only partial privacy.

Acknowledgements

We are grateful to Nir Shavit for pointing out several important applications of incremental cryptography.

References

[AHU] A. Aho, J. Ullman, and J. Hopcroft. The design and analysis of computer algorithms. Addison-Wesley, 1974.

[AKS] M. Ajtai, J. Komlós and E. Szemerédi. An $O(n \log n)$ sorting network. STOC 83.

[Ba] K. Batcher. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference* 32, 1968.

[BGG] M. Bellare, O. Goldreich and S. Goldwasser. Incremental cryptography: The case of hashing and signing. Crypto 94.

[BGR] M. Bellare, R. Guérin and P. Rogaway. XOR MACs: New methods for message authentication using block ciphers. Manuscript, March 1994.

[BKR] M. Bellare, J. Kilian and P. Rogaway. The security of cipher block chaining. Crypto 94.

[BR] M. Bellare and P. Rogaway. Entity authentication and key distribution. Crypto 93.

[CW] L. Carter and M. Wegman. Universal Classes of Hash Functions. *J. Computer and System Sciences* **18**, 143–154, 1979.

[Go] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. STOC 87.

[GGM] O. Goldreich, S. Goldwasser and S. Micali. How to construct random functions. *Journal of the ACM*, Vol. 33, No. 4, 210–217, 1986.

[GM] S. Goldwasser and S. Micali. Probabilistic encryption. *J. of Computer and System Sciences* 28, 270–299, April 1984.

[GMR] S. Goldwasser, S. Micali and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[KR] R. Karp and M. Rabin. Efficient randomized pattern matching algorithms. *IBM J. of Research and Development* Vol. 31, No. 2, March 1987.

[Me1] R. Merkle. A certified digital signature scheme. Crypto 89.

[Me2] R. Merkle. Protocols for public key cryptosystems. *Proceedings of the 1980 Symposium on Security and Privacy.*

[Os] R. Ostrovsky. Efficient Computations on Oblivious RAMs. STOC 90.

[Ri] R. Rivest. The MD5 message-digest algorithm. IETF Network Working Group, RFC 1321, April 1992.

A Sketch of proof of Theorem 3.2

A key observation regarding the incremental tagging algorithm follows.

Proposition A.1 Suppose that Tag is a valid tag-tree for the text T, stored as version $\mathbf{cnt}$ of document (name) α. Then the tag-trees produced by the system in response to a cut-operation, with parameters α, β, γ, are valid tag-trees for the resulting texts (when stored as new versions of documents (names) β and γ). Similarly, for pasting.

We stress that the term "validity" used in the above proposition and below includes the requirement that the document-name and version-counter authenticated by the root match the actual document-name and the corresponding current version-counter.

We show that attacks on our tree tagging system cannot be too successful since they would yield successful attacks on the basic message authentication scheme MA.

Consider an arbitrary adversary that attacks the tagging system using commands of all three types (i.e., 'create', 'modify' and 'tamper'). Note that both tagging algorithms employed (for 'create' and 'modify') use an oracle to MA_a (and VMA_a), for a randomly generated authentication-key a. We assume for simplicity that the adversary always halts outputting a properly tagged document (i.e., a pair (D, μ) where μ is an MA_a-valid authentication tree for D). We stress that this (document,tag)-pair is not necessarily one which has not "appeared before" (i.e., D may have appeared as a previous virtual message). Actually, our task is to show that it is most likely that the document D has appeared previously as a virtual document (see Section 3.1 for terminology).

We now consider two events defined over the probability space of all possible executions of the above attack. The *first* *event* is that the adversary has produced (either as part of a tampering command or as part of its output) a tree-tag containing an MA_a-tag for a string for which an MA_a-tag did not appear as part of some tag-tree created by the system (in response to some 'create' or 'modify' command). The *second event* is that the same MA_a-tag appears as the tag of two different strings in either two different tag-trees or in the same tag-tree, produced by the system (in response to some 'create' or 'modify' command). Both events may occur only with negligible probability, since each of them constitutes a breach of the security of the basic message authentication scheme MA. If none of the events occur, we call the execution *good*.

From this point on, we assume that the execution is good and show that (in this case) the tag-tree output by the adversary is for a document, denoted D, which has appeared before as a virtual document. Since the tag-tree output by the adversary is valid, it follows that all the MA-tags appearing in it are valid. By the assumption that the execution is good, it follows that all these tags, and in particular the tag of the root, have appeared in some previous tag-tree (produced by the system). Consider the earliest time t in which there exists a document name α with a cryptographic form having a vertex v with the same MA_a-tag as the root of D (i.e., the document output by the adversary). Since roots have a special form, the node v must be the root of the tag-tree. Let V_t be the virtual document associated with document-name α at that time t. Note that V_t was defined in time t by either a create or a modification command.

If V_t was defined by a document-creation command then the tag-tree of document α at time t must be valid (as it was produced in response to a create command). By the assumption that the execution is good, it follows that this (valid) tag-tree is identical to the (valid) tag-tree of D (since otherwise two different valid tag-trees, with an identical root, have appeared in the execution implying that the execution contains two different strings with the same MA_a-tag). It follows that $V_t = D$.

We are left with the case where V_t was defined by a document-modification command. For each virtual document, we define a virtual tag-tree (associated with it). The definition mimics the one of a virtual document (i.e., it ignores the possible tampering of the tag-trees associated to document-names). Namely,

- The virtual tag-tree associated with a creation command (and with the virtual document defined by this command) *is the actual tag-tree produced by the system.* Thus, in this case, the virtual tag-tree is a valid tag-tree of the corresponding virtual document.
- The virtual tag-tree associated with a modification command (and with the virtual document defined by this command) *consists of a tree of* MA*-tags in which the new tags* (produced at this stage by the system) *are the actual ones but the tags of the other vertices are as in the virtual tag-tree of the corresponding virtual documents.*

An important observation, proven by induction on the recursive definition of a virtual tag-tree, is that every *virtual* tag-tree consists of MA_a-tags which were produced by the system. Combining the same type of induction with Proposition A.1, we prove the following

Lemma A.2 At any time, the virtual tag-tree associated with each document is a valid tag-tree for the corresponding virtual document.

We stress that the assertion of the lemma does not necessarily hold with respect to the *actual* tag-trees that may even contain illegal MA_a-tags.

Proof: First, we observe that the lemma holds for a virtual tag-tree defined by a create operation. Now, consider a virtual tag-tree defined by pasting documents (names) α and β. By definition, this virtual tag-tree consists of the MA_a-tags of the corresponding virtual tag-trees and the actual tags produced for the vertices along the path from the topological change to the root. We claim that if a path from some vertex in the *actual* tag-tree of α (resp., β) to its root is valid then the labels of the children of the vertices on this path equal the corresponding labels in the *virtual* tree of α (resp., β). Once this claim is proven we are done (since then we are guaranteed that the newly formed MA_a-tags are tags for the correct values).

The claim is proven by induction from the root of this path, using the hypothesis that the execution is good, the fact that the virtual tag-tree of α consists of MA_a-tags produced by the system, and the hypothesis that the virtual tag-tree of α is a valid tag-tree (for the corresponding virtual document). Firstly, if the tag of the root of the actual tag-tree for document α having current version-counter $\mathtt{cnt}$ is valid then it must have been produced by the system for document α at the time its counter was incremented to the value $\mathtt{cnt}$. Thus, the root of the actual tag-tree equals the root of the virtual tag-tree of α. It follows that the labels of the children of root of the actual tag-tree are as in the corresponding virtual tag-tree. In particular, the subtree counters of the corresponding children in the two tag-trees are equal and thus the locations of the corresponding subtexts are the same. Similarly, if the tag of a vertex, v, of the actual tag-tree is both valid and equals the corresponding tag in the virtual tag-tree then the labels of v's children are as in the corresponding virtual tag-tree. The claim follows and so does the lemma. ∎

We now claim that the virtual tag-tree of V_t equals the tag-tree produced by the adversay (for D). First, we observe that the actual tag-tree of V_t at time t contains as its root an MA_a-tag of a string containing the document name α and a counter-version denoted $\mathtt{cnt}$. However, by validity of the virtual tag-tree of V_t it follows that also the virtual tag-tree contains as its root an authentication of the document name α and the counter-version $\mathtt{cnt}$. Combining this with the definition of the incremental algorithm and the fact that each MA-tag in the virtual tag-tree has appeared in an actual tag-tree, we infer that the roots of both the actual and virtual tag-trees of V_t are identical. Thus, the virtual tag-tree of V_t and the output tag-tree of D have identical roots (recall that by definition the root of the actual tag-tree of V_t equals the root of the tag-tree of D). By the assumption that the execution is good and the fact that the virtual tag-tree consists of MA_a-tags produced by the system, we conclude again that these two (valid) tag-trees (ie., the virtual tag-tree of V_t and the tag-tree of D) must be identical, and again $V_t = D$ follows. This concludes the proof that in every good execution the authenticated document output by the adversary (ie., D) is a virtual document which has appeared before.