CryptEmacs: Incremental Cryptography Reduced to Practice

February 16, 1998

Shafi Goldwasser: Laboratory for Computer Science,
MIT and Department of Applied Mathematics and Computer Science,
Weizman Institute of Science, Rehovot, Israel.
e-mail: shafi@theory.lcs.mit.edu

Yoav Yerushalmi: Laboratory for Computer Science,
Massachussetts Institute of Technology,
building NE43-334, 545 Technology Square,
Cambridge, MA 02143, USA.
e-mail: yoav@mit.edu
phone: (617) 253-5866

Contact Author : Yoav Yerushalmi.

CryptEmacs: Incremental Cryptography Reduced to Practice

February 16, 1998

Keywords: Incremental cryptography, Dynamic updates, XOR-MAC.

Abstract

The topic of this paper is a "cryptographic editor" - an editor which updates the value of cryptographic transformations on files being edited as they are edited, optimizing the use of idle CPU time during the session. At any point in time during an editing session, the cryptographic transformations of the current version of the file is readily available.

The use of an incremental algorithm (see [BGG94]) for computing the cryptographic transformation at hand (in our case, a MAC of a file), ensures that the performance of such an editor is not noticeably worse than the underlying standard editor. It gives the guarantee that the cost of updating the value of the cryptographic transformation after every few editing operations is proportional to the the amount of change made to the file and not the entire size of the file.

In this paper we describe the results of an actual implementation of (to our knowledge) the first cryptographic editor. In particular, we incorporate an incremental scheme for generating message authentication codes (MAC) into a commonly used text editor (Emacs), and optimized the code in such a way that the performance of the enhanced editor is, perhaps surprisingly, not noticeably any different than standard Emacs (which does not perform any MAC computations). We also report the conclusions derived from the project that can be applied to future implementations.

1 Introduction

Many, if not most, users today composing and sending e-mail, using the web, saving files, or accessing data bases with sensitive queries, seem to opt for *not* using the many cryptographic inventions available.

Political reasons aside, we have identified several reasons for this:

1. **Performance.** The perception founded in reality is that current schemes take a noticeable amount of time.
2. **Ease of use.** Many of the current schemes and implementations take work to integrate into existing applications, and require the user to run an external program. Users are not willing to explicitly do any extra work to obtain security.

3. **Psychological acceptance.** Many users do not understand cryptography and because of the above overheads, do not bother to take the extra steps necessary for using it.

Thus, a person composing an e-mail message doesn't really care whether the cryptographic algorithm available has polynomial or exponential order of growth, which one is more space-efficient, or who devised the protocol. While all of the above are important, it is actually the end-effect which the user of the scheme desires. They want the scheme not to delay their work, they want the scheme to be simple to use, and they want to believe the scheme is secure.

1.1 Our Approach

The approach we suggest is to embed cryptography directly into existing applications. For example, take a commonly used editor (such as GNU Emacs) and modify it to automatically sign files as you edit them, using CPU time which is normally unused. The cryptographic operations will happen by default and require no action on the part of the user (short of specifying a key at some predetermined point). This idea is, of course, unworkable if the integration of cryptography into the application slows it down in a noticeable fashion or if it requires intensive pre-computation or post-computation.

To make our approach workable, we chose to embed *incremental cryptographic* algorithms into existing applications. The goal being to ensure that the extra work needed to perform cryptographic operations is distributed over the lifetime of the application at hand, in such a way that little (or no) perceivable slowdown occurs at any point in time. *Incrementality* was introduced in [BGG94] as a measure of efficiency in a cryptographic scheme. In [BGG94] an incremental hashing method was proposed, and in [BGG95], several other possible incremental cryptographic schemes were proposed for authentication and encryption, facing adversaries of various strength, e.g. signing documents for virus detection purposes. For example, the idea of an incremental digital signature scheme is that updating the value of a previous signature of text in accordance with changes made to the text, takes work which is proportional to the amount of change made and not the length of the text.

One of the things that incremental cryptography allows is the *dynamic* update of the cipher to reflect changes in the source document. Taken to extreme, if the incremental update procedure is efficient enough, it is possible for the scheme to generate the correct new cipher each time the user types a key and modifies the source. If this works, then when the user is done typing, the document already has its cipher computed, and the user has no need to wait for the encryption code to start and finish working. Namely, the the time spent for cryptographic purposes on "small" changes made by the application is very "short", and can be incorporated into time slices which fit within typical idle times in editing.

Assuming that the above idea works, it is now possible to make whatever scheme is being used (in this case, we will generate message authentication codes) compute for EVERY document being used. Since this can be done in a manner which does not appear to slow the user down it is possible to integrate this into the operating system or text processor and compute this for all documents, including those that the user would normally not bother requesting MACs for. Now, at login time (or when loading the text editor), the user can be prompted for the password he uses once, and then things will be done unobtrusively for the user. In the long run, this will both seem simple AND fast, even though more computations are being performed than would be if the computation was asked for at the end.

1.2 The System

To see if this idea was actually viable, we implemented the XOR scheme as proposed in [BGG95] using elisp, and incorporated it into Emacs as a minor mode (so it can coexist with other Emacs modes). We then tried out the scheme to see how well it fared as compared to using nothing, and compared to generating MACs at the end.

It must be pointed out here that the difficulty in implementing the system did not lie in the complexity of the algorithm. It was a fairly simple proven XOR and DES combo. However, the challenge lay in integrating this concept into an editor in such a way as to not slow down the user, and at the same time also have a secure computation ready at the end. At this point, tying the computation into the editor, and making full use of the underlying operating system features are critical. One has to be able to identify idle CPU time and use it to the maximum.

After some idea tossing, we decided to test the performance of the system using the amount of time it took for the final computation to be prepared from the moment it was requested, with the added requirement that whatever else happens in the system, the user should not notice the computations being performed in the background. Using this measurement, we ended up with a system whose efficiency was considerably better than any other systems used today, and whose efficiency can be increased considerably with extra knowledge of the kind of edits being made.

We named the system CryptEmacs, to reflect the nature of encryption placed within the framework of Emacs. To summarize the system's properties:

1. CryptEmacs is easy to use. When installed, the usage of CryptEmacs is exactly the same as the use of Emacs. The only difference that may be noticed is the creation of '.sum' files containing MAC data for files.

2. CryptEmacs is fast. Editing documents within it appears to take the same amount of time as editing in Emacs (with no security options) for most users.

3. CryptEmacs supports all the popular modes of Emacs, which include e-mail handling clients, text editor, lisp, c, and java code editor, www mode, etc. The implementation makes it work alongside all current Emacs modes.

4. CryptEmacs is designed to work with any type of key management system offered by the OS or the administrator of the system.

5. Verification of MACs produced by CryptEmacs take as long as generation of the MAC from scratch, which has already been argued in previous papers to be comparable in time to non-incremental schemes in use today.

It is interesting to note that some of the system implementation issues to be addressed are similar to (1) those addressed by a system implementing garbage collection (a task which need be done in the background while ordinary computations are taking place and for which the choice of when its done both depends on the changes made to files and idle times); and (2) those that need be addressed by backup systems which must periodically save files or changes made to them to ensure that computer crashes do not imply massive loss of work.

We note that in common use of cryptographic protocols, there is quite a bit of systems work to be done in the design of protocols. We believe that this paper is the one of the first examples of a system which take into account system properties when designing a cryptographic algorithm.

Road map to this paper: In section 2 we provide the previously supplied definitions for incremental MAC and in section 3 describe in detail the particular XOR scheme outlined in [BGG95] which we incorporated into Emacs. In section 4 we describe the implementation and its performance. We conclude by suggesting other directions which this and other incremental schemes may proceed to.

2 Incremental Cryptography in the Case of MACs

2.1 Symbols

In the definitions in this paper, we use the following notations:

1. A document D is viewed as a sequence of blocks, each b long, and we let $B_b = 0, 1^b$ be the domain for the blocks.
2. A document has n blocks, making B_b^n the space of an n-block message.
3. $D[i]$ is the i-th block of $D \in B_b^n$.
4. IncM is the incremental scheme being used to update the MAC of the document D.
5. A change in the document is denoted by the following requests:
   - A replacement is denoted by $\mathrm{IncM}(M, D, mac, \mathrm{repl}, (j, m))$. In this request, $D[j]$ will now contain m and the rest of D will remain unchanged. We let $D\langle j, m\rangle$ be shorthand for the above, where m is the new block.
   - A deletion request is denoted by $\mathrm{IncM}(M, D, mac, \mathrm{del}, j)$. In this request, $D[j]$ is removed from the sequence, so $D[j-1]$ is followed by $D[j+1]$ in the document. M is also modified to reflect the shorter document. Stated otherwise, the new document now looks like:

$$D = \begin{cases} (D[1]..D[n]) & \text{if } j = 0 \\ (D[0]..D[j-1] \cdot D[j+1]..D[n]) & \text{if } 0 < j < n \\ (D[0]..D[n-1]) & \text{if } j = n \end{cases}$$

$$n = n - 1$$

     shorthand for the above is $D\langle -j\rangle$.
   - An insertion request is denoted by $\mathrm{IncM}(M, D, mac, \mathrm{ins}, (j, m))$. In this request, the j-th block is now followed by a new block containing m. Also, M is modified to reflect the new length. Otherwise stated:

$$D = \begin{cases} (m \cdot D[0]..D[n]) & \text{if } j = -1. \\ (D[0]..D[j] \cdot m \cdot D[j+1]..D[n]) & \text{if } -1 < j < n. \\ (D[0]..D[n] \cdot m) & \text{if } j = n. \end{cases}$$

$$n = n + 1$$

shorthand for the above is $D\langle +j, m\rangle$.

2.2 Incremental MACs

We first need to extend the definition of a message authentication code to allow for incrementality. We introduce independence (as suggested in [BGG94]) of the security parameter k, the number of blocks in the message b, and the size of each block n.

Definition 2.1: a family of message authentication code computing functions is defined by the pair $\mathcal{M}$ = (Mgen, Meval) of algorithms.

- The PPT generator Mgen takes as input 1^k, 1^b, 1^n, and returns a string M.
- The PPT evaluator Meval takes M and a document $D \in B_b^n$, and outputs a k bit string that is the message authentication code for the appropriate document.

We now need to create an update function that will allow us to incrementally change the MAC without recomputing from scratch. This is achieved via IncM which turns the MAC of D into the MAC of $D\langle j, m\rangle$, $D\langle -j\rangle$, or $D\langle +j, m\rangle$ depending on the change desired. We use ideas presented in previous papers to extend MAC computing functions.

Definition 2.2: Let $\mathcal{M}$ = (Mgen, Meval) specify a family of MAC computing functions. We say that IncM is an update algorithm for $\mathcal{M}$ with running time $T(\cdot,\cdot,\cdot)$ if

$$\forall k, b, n, \forall M \in [\text{MGen}(1^k, 1^b, 1^n)]\ \forall j \in \{1, ..., n\},\ \forall m \in B_b,$$

if mac = MEval(M, D) then it is the case that:

- IncM$(M, D, mac, \text{repl}, (j, m))$ halts in $T(k, b, n)$ steps with an output equivalent to MEval$(M, D\langle j, m\rangle)$.
- IncM$(M, D, mac, \text{del}, j)$ halts in $T(k, b, n-1)$ steps with an output equivalent to MEval$(M, D\langle -j\rangle)$.
- IncM$(M, D, mac, \text{ins}, (j, m))$ halts in $T(k, b, n+1)$ steps with an output equivalent to MEval$(M, D\langle +j, m\rangle)$.

We call the IncM-*augmentation* of $\mathcal{M}$ =(Mgen, Meval) the triple $\mathcal{M}^+$ =(Mgen, Meval, IncM).

3 The XOR Scheme

We now analyze a scheme that implements a message authentication code for documents (originally proposed in [BGG95]). And which has the listed properties of incrementality with a running time $T()$ that is intended to be efficient enough for dynamic updating of the document.

3.1 Initial computation

The scheme proposed in [BGG95] works in the following manner:

There is a key $K = (k_1, k_2)$ which is held by both the MAC generator and MAC verifier in secret. Furthermore, there is a pad-generating function `rand` which adds a randomizer to every block in the message (i.e. given a string σ, it returns $\sigma \cdot r$ where r is a random value). There are two encoding functions, f_1 and f_2, which take as indexes (keys) k_1, k_2 respectively and are used to compute the MAC:

To compute the MAC for message $D = (D[1]..D[n])$ we prefix it with a special start block $D[0]$ and postfix with an end block $D[n+1]$, yielding $D = (D[0]..D[n+1])$. Then, for each block, we add a randomizing pad by calling `rand` with the value in each respective block. This yields a series of $n+2$ blocks containing the data and random pad for each block: $R = (R[0]..R[n+1])$.

Now, we use an idea proposed in [BGR94] for generating MACs, which is to block-cipher-chain the respective R's in the following manner:

$$mac = f_2(\bigoplus_{i=0}^{n} f_1(R[i], R[i+1]))$$

This is the initial MAC, and from this point on, all changes to it are computed incrementally depending on the type of change. Note that an incremental computation in this case yields a MAC that is exactly the same as computing it again from scratch (i.e. there is no history of changes encoded in the MAC), this therefore achieves the requirements of *perfect privacy* as defined in earlier papers.

3.2 Incremental computations

There are two approaches to computing the new MAC. Method one assumes that space is not a concern and saves all subcomputations for the document. Method two trades a small amount of computation time for space-efficiency, and only stores the random pads and the final MAC. Method two, however, also requires that f_2 be reversible, which may or may not be problematic (in our implementation, it was not, since f_2 was reversible anyway). Here are the ways to deal with changes to the document under both models:

3.2.1 Modifying a block

A change in the data of only one block is easy to cater for. $D\langle j, m\rangle$'s MAC is computed by the following technique:

1. Under this method, all the block pairs already have their f_1's computed, so all that is needed is to find the two blocks whose f_1's are affected (block pairs $(D[j-1], D[j])$ and $(D[j], D[j+1])$) , and update them. Then, recompute the final sum.
2. If we do not have the local computations stored somewhere, then we need f_2 to be reversible, and we do the following:

$$\begin{aligned} mac = f_2\Big(& f_2^{-1}(mac) \oplus f_1\big(D[j-1], D[j]\big) \oplus f_1\big(D[j], D[j+1]\big) \\ & \oplus f_1\big(D[j-1], \mathtt{rand}(m)\big) \oplus f_1\big(\mathtt{rand}(m), D[j+1]\big)\Big) \end{aligned}$$

3.2.2 Adding a block

$D\langle +j, n\rangle$'s new MAC is computed thus:

1. Under this scheme we compute the two new f_1's, and use the rest of the computations already performed to get a new hash which we call f_2 upon. The two new block-pairs are respectively $(D[j], m)$ and $(m, D[j+1])$.
2. For this scheme, again we perform some more work, and the formula we use is:

$$mac = f_2\Big(f_2^{-1}(mac) \oplus f_1\big(D[j], D[j+1]\big) \oplus f_1\big(D[j], \mathtt{rand}(m)\big) \oplus f_1\big(\mathtt{rand}(m), D[j+1]\big)\Big)$$

3.2.3 Deleting a block

The final type of change that can be made is a deletion of a block. To compute the MAC for $D\langle -j\rangle$, the following is done:

1. For a scheme where everything is stored, we just need to recompute one pair, the $(D[j-1], D[j+1])$ pair, and throw away two old ones.
2. For the less memory-intensive scheme, we compute the following:

$$mac = f_2\Big(f_2^{-1}(mac) \oplus f_1(D[j-1], D[j]) \oplus f_1\big(D[j], D[j+1]\big) \oplus f_1\big(D[j-1], D[j+1]\big)\Big)$$

4 The Implementation

In order to make this as unobtrusive and as automated as possible, we chose to integrate the MAC computing function directly into some commonly used word-processor/text-editor. We chose Emacs for this purpose since Emacs has a built-in language that is portable and powerful (elisp).

We created a minor mode in Emacs (MAC-mode) which performed the necessary computations per-buffer. Being a minor mode, it works in coalition with other modes, so it does not present any unexpected behavior or incompatibilities (a requirement should this mode be used for everything that is done within Emacs). This of course means that MACs can be generated for everything, from e-mail messages composed in Emacs mh-mail, to C++ code, to buffer edits, and these MACs are computed dynamically in the background.

The code to achieve this, along with documentation, can be found at (see footnote)[1]

[1] Due to the anonymous nature of the submission, it is impossible to include the URL here. However, should the committee accept this paper (or request the information), this text will be replaced by the appropriate URL.

4.1 Internal Representation of Data

When a document is first loaded (or `MAC-mode` is activated on a buffer, there is a check to see whether the data representing the buffer (including the computation history) is available in an auxiliary file (saved as {ffilename}.sav). If it is there, it is loaded and used, otherwise, the basic data structure is created thus:

- First, it breaks the buffer (usually an empty one) into blocks of size `MAC-blocksize`. This value can be set anywhere from 1, which indicates 1 byte, to however large the largest string Emacs will allow is (usually the size of a page in the operating system).
- Next, it generates a pad of size `MAC-security-padding` bytes for each block. The same restrictions apply as to the data in each block.
- It marks every f_1 value in the blocks as `'nil`, which is reserved to indicate that the checksum has not been computed yet.
- Finally, it adds an end-block to the linked list, which is used to indicate there are no more blocks in the list.

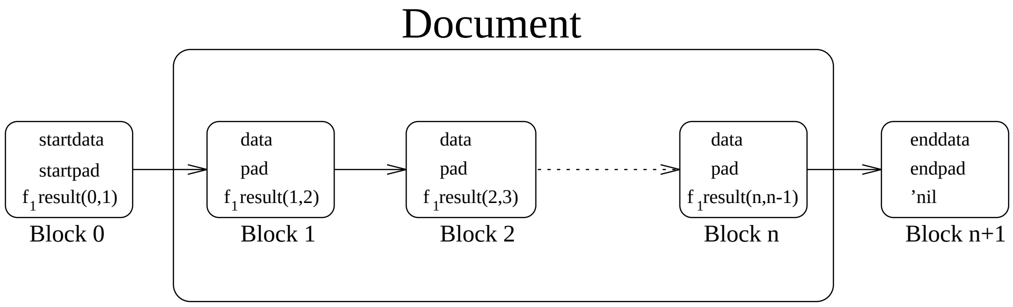

Figure 1: MAC-list data structure

The above steps form the *MAC-list* data structure (see Figure 1). Elisp linked-list (car/cdr pairs) with the properties that the `MAC-string` element for each item in the list holds a `MAC-blocksize`-length string which reflects the contents of the buffer. If all the `MAC-strings` were laid end-to-end, the exact contents in the buffer will result. The `MAC-pad` holds the random pad (that which is gotten from the `rand()` function. Finally, the `MAC-sum` holds the result of the f_1 function called upon the `MAC-string` and `MAC-pad` of the current block and of the next block. It may also hold the special value of `'nil` to indicate that the computation has not been performed yet.

The above implies that the end-block's sum is always 'nil, and that the start-block's sum is variable (so it cannot be made into a constant, and is a different data-block for each buffer). Also, there is one further problem, which lies in the fact that the last data block may not always contain exactly `MAC-blocksize` bytes of data. This is acceptable from a security point of view, but in the implementation, this is restricted to happening on the last block only. (See the *Changes* section for a discussion of variable sized blocks).

4.2 Operation of the Editor

Now, we have a data structure to represent the buffer. From this point on, one of two things may happen: Either the machine will be idle, or the user will be typing something. If the machine is idle, the mode will attempt to find f_1 values that have not been computed. If the user is typing, then it will attempt to make the *MAC-list* data structure reflect the status of the buffer.

If the machine is idle, then the minor mode will utilize this time to update the MAC dynamically. Since most users take breaks while typing, and since most users don't type that quickly anyway, most of the CPU time is spent in this mode. During this time, the list is traversed until a block is found whose `MAC-sum` is set to `'nil`. Once a block is found that has this property, its sum is computed based on the data stored within the current as well as the next block. If all blocks are computed correctly, then the buffer's MAC is computed and held until the time when the user requests it, or else the buffer changes again. At this point, the machine will become completely idle, unless it is also running other processes.

If instead of leaving the machine idle, the user types something which causes changes to the buffer, the internal *MAC-list* is changed to reflect those changes. There are many possible types of changes, and since the algorithm is designed to work in blocks, each of these changes can lead to several possible types of changes in the structure. Any change is reported as a $(startpos, endpos, newlen)$ triplet, which is all that is necessary to figure out what has changed:

- A modification usually occurs when in *overwrite* mode. In this mode, anything that is typed is typed over previous characters. It can, however, also occur in some specific modes where some text that is being typed is replaced with different text (for example, automatic capitalization). In this case, no length change occurs ($endpos - startpos = newlen$). This of course means that the *MAC-list* structure doesn't change, although the data within the respective blocks, as well as the respective f_1 values, do. This is easy to deal with.

- A deletion can occur due to a delete or backspace key. Or on a larger scale, due to a cut operation (among others). This can be more tricky, as the deletion modifies the length of data in a block, and in some cases, can delete an entire block or more. The implementation can only use the algorithm's delete operation when an entire block is removed. In all other cases, the length of a block changes, and so in the worst case, characters from further blocks need to be shifted into the current block to fill it to the right length (and so on for the further ones down). Deletions early on in a document can lead to the entire document's MAC being recomputed from scratch.

- Insertions are very similar to deletions. They can be brought about by almost any editing command in Emacs, as well as automatically due to things like C-mode. Like deletions, they cause block data lengths to change, and so require either pushing data forward through blocks, or in lucky cases, the new data fits completely into a new block between two other ones.

If any block changes, its data is set to reflect the new data, but the sum is left as a `'nil`. The cryptographic computations are not done until such time as the user isn't typing anything, or the user forces the computations (for example, by asking for the MAC, or saving out the file).

An attempt to save the file causes the editor to go through the entire structure to make sure it is correct (all data blocks contain the right number of bytes of data from the buffer, and all f_1 computations have been performed). Then it computes the xor of all the f_1 values (if that has not yet been done in idle time), and finally, computes the f_2 of the xor. The *MAC-list* data structure, as well as the result of the f_2 (the MAC) are written out to the filename with a '.sum' postfix.

4.3 Analysis of Performance

The above scheme was implemented, and tested using a pentium II-300 machine running NetBSD/i386. The Emacs used was Emacs 20.2. For measurement purposes, we set the f_1 and f_2 to be DES, although there are any other symmetric cipher would have worked as well. The typing rate of the person using the software was approximately 60 words per minute.

While most cryptographic analysis tends to focus on order of growth of algorithms, or on the running time of the program, the kind of analysis that is performed for incremental cryptography, especially when used in the kind of scenario described within this paper, needs to focus on perceived speed.

Unfortunately, perceived speed isn't measurable in seconds, but can be thought of as the amount of his time a user feels is being taken up by this additional feature. Therefore, a program that takes five minutes to do something, but does it when the user is taking a break, will feel faster to the user than a program which only takes a minute to do its work, but does it while the user is waiting to run the next application.

It is this perceived speed that dynamic recomputation using incremental cryptography is attempting to reduce, and after trying it out, appears to do its work well. The first implementation used DES in elisp[2]. After testing the implementation, it became very clear that the amount of time taken to compute the f_1 function within Emacs was too long, as it couldn't compute for most of the buffer by the time input was completed. It is important to stress here that Emacs is generally slow when dealing with large amounts of data, and is not particularly fast at handling bits within data (which is required for DES).

The elisp DES code was replaced with C code tied in asynchronously, and the computation speed change was overwhelming. Casual editing of buffers yielded an almost instantaneous MAC. The worst case scenario (that of an insertion right at the beginning which was smaller than a blocksize) was slightly slower than computing from scratch, but the average case (that of editing in the middle) was very fast. Even better, since most editing of documents tends to occur at the end (as the document is being composed), changes at the end needed almost no recomputation, and appeared to yield sums instantaneously.

A basic comparison of apparent performance was done, comparing the integrated MAC evaluator with differing parameters to a MAC computation from scratch on the entire buffer. Instead of measuring using CPU time (which is not a useful quantity when discussing an incremental scheme), we measure time at the point the user requests the computation. We then compute a ratio between the amount of time it takes to perform the computation on the entire buffer as opposed to the time it takes to finish the computation using an incremental scheme:

$$\text{performance} = \frac{T(\text{compute for entire buffer})}{T(\text{incrementally compute when done editing})}$$

[2]edes.el. Mark Eichin, Cygnus Solutions, beta software

Figures 2 and 3 detail the performance on the machine we tested on. For the append, a buffer was simply typed into continuously at a the rate specified. For the random edits, cuts, pastes, overwrites, insertions, and other operations were performed on the buffer. It becomes clear that this measure is very dependent on the exact types of changes, and so being able to choose the right constant is very important.

parameters	chars per second		
	1	4	10
blocksize = 1 bit pad size = 1 bit	4	3	1
blocksize = 56 bits pad size = 8 bits	∞	∞	11

Figure 2: Editor performance on append operations

parameters	words per minute		
	10	30	60
blocksize = 1 bit pad size = 1 bit	4	4	3
blocksize = 56 bits pad size = 8 bits	8	6	1

Figure 3: Editor performance on random access edits

As can be seen in the above data, the kind of input, as well as the person who is typing, and the choice of variables, all control the efficiency of the scheme. Furthermore, since this scheme is implemented in elisp, it can be improved upon considerably by integrating it using C into a different word processor (the optimization we used only moved the DES operation into C – further tests with moving the entire data structure into a C subprogram can improve performance even more). While it may seem that only order of growth of the algorithm is important, it turns out that small optimizations can make major differences in how well the scheme inter-operates with the user.

One final thing that deserves some analysis is the verification scheme. Theoretically, verification should be approximately as fast as generation from scratch, and in practice, it is. Incrementality as of yet does not offer any speedups in the verification scheme (although should one choose to use diffs, they are smaller to verify).

5 Other Ideas That Emerged From the Project

After trying it out for a while, we had noticed several interesting problems, and had alternative approaches to solving them.

- The constant blocksize (initially envisioned to be 1 byte by the authors) can lead to several slow-downs. If it is set at just one character, then there are a lot of f_1's computed per document. If it is set at a large number, then the probability of causing incomplete blocks in the middle of a document are increased (leading to the need to adjust data further down, and invalidating all the f_1's further down the list). It turns out that depending on the kind of editing that is done more often, a different model should be used. Larger block-sizes lend themselves well to append operations, and database-style operations that dealt with data in blocks. One byte block-sizes were better for random-access edits.

- Another idea noticed was that there really was no need for a constant blocksize. Allowing the size of the data in a block to vary stops all the problems caused by data being unaligned. Since nothing in the proof of security contains anything requiring the blocks to use constant lengths, we can allow for blocks to have varying sized data.

- Although this was tied into Emacs in the hopes of making it all-encompassing for file modifications, it is also instead possible (and somewhat more appropriate) to integrate the MAC scheme into the filesystem. This will get and generate 'checksums' for every file, allowing for protections from viruses as well as detecting unauthorized changes. Since most changes to files on a filesystem tend to be of the 'insert, delete, overwrite, and append', it can easily map to the same ideas as were used for the documents. Furthermore, a lot of files on the machine which are in constant change tend to be log files, and those are handled via append operations only, which is ideal for the proposed MAC generating scheme. Study of LFS suggests that operating in terms of changes to files is a worthwhile approach in many cases [RO90].

- Other ideas for uses of this scheme include the concept of submitting "diff's" instead of documents. Some schemes, such as revision control (through RCS / CVS) lend themselves well to incremental cryptography, since they already care only for changes to the file, instead of the file itself. Since the data exchange is now based on changes to the data, it is very easy to make the incremental scheme part of the overall system.

- MACs are not the only things a user might want for his files. Signatures and encryption are also very useful, and integrating and testing those into commonly used applications may be worthwhile. The authors are pursuing this venue right now.

- These ideas can also be merged with predictive/adaptive algorithms to allow the computer to try and *guess* what the user will type, and pre-compute some blocks.

6 Conclusion

Incremental Cryptography lends itself very well to solving the problem of "Why aren't more people using cryptography". It makes schemes appear fast, thereby making it more natural to integrate those schemes into existing applications. With that done, since slowdown does not appear to be an issue, it makes sense to have these schemes operate for EVERY document.

Our implementation showed that this idea is viable, and has suggested the kind of problems that may not be immediately apparent in the real world. Ultimately, it would be nice to assume that all of this will be part of the operating system or the applications running, and integrated unobtrusively, so a user will no longer have to choose to sign his mail or encrypt his file. This will occur automatically based on a single password provided at login time.

References

[BGG94] M. Bellare, O. Goldreich, and S. Goldwasser: "Incremental Cryptography: The case of Hashing and Signing." Crypto'94.

[BGG95] M. Bellare, O. Goldreich, and S. Goldwasser: "Incremental Cryptography and Application to Virus Protection." STOC'95.

[BGR94] M. Bellare, R. Guérin, and P. Rogaway. "XOR MACs: New methods for message authentication using block ciphers". manuscript March 1994.

[GNU] GNU Manual Group, "The GNU Emacs Lisp Reference Manual".

[Mi96] D. Micciancio: "Oblivious Data Structures: Applications to Cryptography", (manuscript).

[RO90] M. Rosenblum, J. Ousterhout, "The Design and Implementation of a Log-Structured File System" ACM Transactions on Computer Systems 10, 1 (Feb. 1992), 26-52.