discarded to save disk space.
To facilitate this without
breaking the block's hash,
transactions are hashed in a
Merkle Tree [7][2][5], with
only the root included in the
block's hash. Old blocks can
then be compacted by stubbing
off branches of the tree. The
interior hashes do not need
.to be stored
Block
Hash0 Hash1 Hash2 Hash3
Tx0 Tx1 Tx2 Tx3
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Block
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Hash2 Hash3 Tx3
Transactions Hashed in a
Merkle Tree After Pruning
Tx0-2 from the Block
A block header with no
transactions would be about
80 bytes. If we suppose
blocks are generated every 10
minutes, 80 bytes * 6 * 24 *
365 = 4.2MB per year. With
computer systems typically
selling with 2GB of RAM as of
2008, and Moore's Law
predicting current growth of
1.2GB per year, storage
should not be a problem even
if the block headers must be
.kept in memory
4
Simplified Payment .8
Verification
It is possible to verify
payments without running a
full network node. A user
only needs to keep a copy of
the block headers of the
longest proof-of-work chain,
which he can get by querying
network nodes until he's
convinced he has the longest
chain, and obtain the Merkle
branch linking the
transaction to the block it's
timestamped in. He can't
check the transaction for
himself, but by linking it to
a place in the chain, he can
see that a network node has
accepted it, and blocks added
after it further confirm the
.network has accepted it
Longest Proof-of-Work Chain
Block Header
Block Header
Prev Hash
Nonce
Prev Hash
Nonce
Prev Hash
Nonce
Merkle Root
Merkle Root
Hash01 Hash23
Transaction
In Out
... ... In
Hash2 Hash3 Tx3
As such, the verification is
reliable as long as honest
nodes control the network,
but is more vulnerable if the
network is overpowered by an
attacker. While network nodes
can verify transactions for
themselves, the simplified
method can be fooled by an
attacker's fabricated
transactions for as long as
the attacker can continue to
overpower the network. One
strategy to protect against
this would be to accept
alerts from network nodes
0

How to Time-Stamp a Digital Document*

Stuart Haber
stuart@bellcore.com

W. Scott Stornetta
stornetta@bellcore.com

Bellcore
445 South Street
Morristown, N.J. 07960-1910

Abstract

The prospect of a world in which all text, audio, picture, and video documents are in digital form on easily modifiable media raises the issue of how to certify when a document was created or last changed. The problem is to time-stamp the data, not the medium. We propose computationally practical procedures for digital time-stamping of such documents so that it is infeasible for a user either to back-date or to forward-date his document, even with the collusion of a time-stamping service. Our procedures maintain complete privacy of the documents themselves, and require no record-keeping by the time-stamping service.

*Appeared, with minor editorial changes, in *Journal of Cryptology*, Vol. 3, No. 2, pp. 99–111, 1991.

Time's glory is to calm contending kings,
To unmask falsehood, and bring truth to light,
To stamp the seal of time in aged things,
To wake the morn, and sentinel the night,
To wrong the wronger till he render right.

The Rape of Lucrece, l. 941

1 Introduction

In many situations there is a need to certify the date a document was created or last modified. For example, in intellectual property matters, it is sometimes crucial to verify the date an inventor first put in writing a patentable idea, in order to establish its precedence over competing claims.

One accepted procedure for time-stamping a scientific idea involves daily notations of one's work in a lab notebook. The dated entries are entered one after another in the notebook, with no pages left blank. The sequentially numbered, sewn-in pages of the notebook make it difficult to tamper with the record without leaving telltale signs. If the notebook is then stamped on a regular basis by a notary public or reviewed and signed by a company manager, the validity of the claim is further enhanced. If the precedence of the inventor's ideas is later challenged, both the physical evidence of the notebook and the established procedure serve to substantiate the inventor's claims of having had the ideas on or before a given date.

There are other methods of time-stamping. For example, one can mail a letter to oneself and leave it unopened. This ensures that the enclosed letter was created before the time postmarked on the envelope. Businesses incorporate more elaborate procedures into their regular order of business to enhance the credibility of their internal documents, should they be challenged at a later date. For example, these methods may ensure that the records are handled by more than one person, so that any tampering with a document by one person will be detected by another. But all these methods rest on two assumptions. First, the records can be examined for telltale signs of tampering. Second, there is another party that views the document whose integrity or impartiality is seen as vouchsafing the claim.

We believe these assumptions are called into serious question for the case of documents created and preserved exclusively in digital form. This is because electronic digital documents are so easy to tamper with, and the change needn't leave any telltale sign on the physical medium. What is needed is a method of time-stamping digital documents with the following two properties. First, one must find a way to time-stamp the data itself, without any reliance on the characteristics of the medium on which the data appears, so that it is impossible to change even one bit of the document without the change being apparent. Second, it should be impossible to stamp a document with a time and date different from the actual one.

The purpose of this paper is to introduce a mathematically sound and computationally practical solution to the time-stamping problem. In the sections that follow, we first consider a naive solution to the problem, the digital safety deposit box. This serves the pedagogical purpose of highlighting additional difficulties associated with digital time-stamping beyond those found in conventional methods of time-stamping. Successive improvements to this naive solution finally lead to practical

ways to implement digital time-stamping.

2 The Setting

The setting for our problem is a distributed network of users, perhaps representing individuals, different companies, or divisions within a company; we will refer to the users as *clients*. Each client has a unique identification number.

A solution to the time-stamping problem may have several parts. There is a procedure that is performed immediately when a client desires to have a document time-stamped. There should be a method for the client to verify that this procedure has been correctly performed. There should also be a procedure for meeting a third party's challenge to the validity of a document's time-stamp.

As with any cryptographic problem, it is a delicate matter to characterize precisely the security achieved by a time-stamping scheme. A good solution to the time-stamping problem is one for which, under reasonable assumptions about the computational abilities of the users of the scheme and about the complexity of a computational problem, and possibly about the trustworthiness of the users, it is difficult or impossible to produce false time-stamps. Naturally, the weaker the assumptions needed, the better.

3 A Naive Solution

A naive solution, a "digital safety-deposit box," could work as follows. Whenever a client has a document to be time-stamped, he or she transmits the document to a time-stamping service (TSS). The service records the date and time the document was received and retains a copy of the document for safe-keeping. If the integrity of the client's document is ever challenged, it can be compared to the copy stored by the TSS. If they are identical, this is evidence that the document has not been tampered with after the date contained in the TSS records. This procedure does in fact meet the central requirement for the time-stamping of a digital document.[1] However, this approach raises several concerns:

Privacy This method compromises the privacy of the document in two ways: a third party could eavesdrop while the document is being transmitted, and after transmission it is available indefinitely to the TSS itself. Thus the client has to worry not only about the security of documents it keeps under its direct control, but also about the security of its documents at the TSS.

Bandwidth and storage Both the amount of time required to send a document for time-stamping and the amount of storage required at the TSS depend on the length of the document to be time-stamped. Thus the time and expense required to time-stamp a large document might be prohibitive.

Incompetence The TSS copy of the document could be corrupted in transmission to the TSS, it could be incorrectly time-stamped when it arrives at the TSS, or it could become corrupted

[1] The authors recently learned of a similar proposal sketched by Kanare [14].

or lost altogether at any time while it is stored at the TSS. Any of these occurences would invalidate the client's time-stamping claim.

Trust The fundamental problem remains: nothing in this scheme prevents the TSS from colluding with a client in order to claim to have time-stamped a document for a date and time different from the actual one.

In the next section we describe a solution that addresses the first three concerns listed above. The final issue, trust, will be handled separately and at greater length in the following section.

4 A Trusted Time-Stamping Service

In this section we assume that the TSS is trusted, and describe two improvements on the naive solution above.

4.1 Hash

Our first simplification is to make use of a family of cryptographically secure *collision-free hash functions*. This is a family of functions $h : \{0,1\}^* \to \{0,1\}^l$ compressing bit-strings of arbitrary length to bit-strings of a fixed length l, with the following properties:

1. The functions h are easy to compute, and it is easy to pick a member of the family at random.
2. It is computationally infeasible, given one of these functions h, to find a pair of distinct strings x, x' satisfying $h(x) = h(x')$. (Such a pair is called a *collision* for h.)

The practical importance of such functions has been known for some time, and researchers have used them in a number of schemes; see, for example, [7, 15, 16]. Damgård gave the first formal definition, and a constructive proof of their existence, on the assumption that there exist one-way "claw-free" permutations [4]. For this, any "one-way group action" is sufficient [3].

Naor and Yung defined the similar notion of "universal one-way hash functions," which satisfy, in place of the second condition above, the slightly weaker requirement that it be computationally infeasible, given a string x, to compute another string $x' \neq x$ satisfying $h(x) = h(x')$ for a randomly chosen h. They were able to construct such functions on the assumption that there exist one-to-one one-way functions [17]. Rompel has recently shown that such functions exist if there exist one-way functions at all [20]. See §6.3 below for a discussion of the differences between these two sorts of cryptographic hash functions.

There are practical implementations of hash functions, for example that of Rivest [19], which seem to be reasonably secure.

We will use the hash functions as follows. Instead of transmitting his document x to the TSS, a client will send its hash value $h(x) = y$ instead. For the purposes of authentication, time-stamping y is equivalent to time-stamping x. This greatly reduces the bandwidth problem and the storage requirements, and solves the privacy issue as well. Depending on the design goals for an implementation of time-stamping, there may be a single hash function used by everybody, or different hash functions for different users.

it to generate new coins. He
ought to find it more
profitable to play by the
rules, such rules that favour
him with more new coins than
everyone else combined, than
to undermine the system and
the validity of his own
.wealth
Reclaiming Disk Space .7
Once the latest transaction
in a coin is buried under
enough blocks, the spent
transactions before it can be
discarded to save disk space.
To facilitate this without
breaking the block's hash,
transactions are hashed in a
Merkle Tree [7][2][5], with
only the root included in the
block's hash. Old blocks can
then be compacted by stubbing
off branches of the tree. The
interior hashes do not need
.to be stored
Block
Hash0 Hash1 Hash2 Hash3
Tx0 Tx1 Tx2 Tx3
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Block
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Hash2 Hash3 Tx3
Transactions Hashed in a
Merkle Tree After Pruning
Tx0-2 from the Block
A block header with no
transactions would be about
80 bytes. If we suppose
blocks are generated every 10
minutes, 80 bytes * 6 * 24 *
365 = 4.2MB per year. With
computer systems typically
selling with 2GB of RAM as of
2008, and Moore's Law
predicting current growth of
1.2GB per year, storage
should not be a problem even
if the block headers must be
.kept in memory
4
Simplified Payment .8
Verification
It is possible to verify
payments without running a
full network node. A user
only needs to keep a copy of
the block headers of the
longest proof-of-work chain,
which he can get by querying
network nodes until he's
convinced he has the longest
chain, and obtain the Merkle
branch linking the
transaction to the block he's
timestamped in. He can't
check the transaction for
himself, but by linking it to
a place in the chain, he can
see that a network node has
accepted it and blocks added
after it further confirm the
.network has accepted it
Longest Proof-of-Work Chain
Block Header
Block Header
Prev Hash
Nonce
Prev Hash
Nonce
Prev Hash
Nonce
Merkle Root
Merkle Root
Hash01 Hash23
Transaction
In Out
... ... In
Hash2 Hash3 Tx3
As such, the verification is

For the rest of this paper, we will speak of time-stamping hash values y—random-appearing bit-strings of a fixed length. Part of the procedure for validating a time-stamp will be to produce the pre-image document x that satisfies $f(x) = y$. Inability to produce such an x invalidates the putative time-stamp.

4.2 Signature

The second improvement makes use of digital signatures. Informally, a *signature scheme* is an algorithm for a party, the signer, to tag messages in a way that uniquely identifies the signer. Digital signatures were proposed by Rabin and by Diffie and Hellman [18, 7]. After a long sequence of papers by many authors, Rompel [17] showed that the existence of one-way functions can be used in order to design a signature scheme satisfying the very strong notion of security that was first defined by Goldwasser, Micali, and Rivest [10].

With a secure signature scheme available, when the TSS receives the hash value, it appends the date and time, then signs this combined document and sends it to the client. By checking the signature, the client is assured that the TSS actually did process the request, that the hash was correctly received, and that the correct time is included. This takes care of the problem of present and future incompetence on the part of the TSS, and it completely eliminates the need for the TSS to store records.

5 Two Time-Stamping Schemes

Sed quis custodiet ipsos Custodes?

Juvenal, c. 100 A.D.

But who will guard the guards themselves?

What we have described so far is, we believe, a practical method for time-stamping digital documents of arbitrary length. However, neither the signature nor the use of hash functions in any way prevents a time-stamping service from issuing a false time-stamp. Ideally, we would like a mechanism which guarantees that no matter how unscrupulous the TSS is, the times it certifies will always be the correct ones, and that it will be unable to issue incorrect time-stamps even if it tries to.

It may seem difficult to specify a time-stamping procedure so as to make it impossible to produce fake time-stamps. After all, if the output of a client who has given as input a document x and some timing information τ, is a bit-string $c = A(x, \tau)$ that stands as a legitimate time-stamp for x, what is to prevent a forger some time later from simply using the same timing information τ and then running A to produce the same certificate c? The question is relevant even if A is a probabilistic algorithm.

Our task may be seen as the problem of simulation of a trusted TSS, in the absence of generally trusted parties. There are two rather different approaches we might take, and each one leads to a solution. The first approach is to construct a centralized but possibly untrustworthy TSS to produce genuine time-stamps, in such a way that fake ones are difficult to produce. The second approach is somehow to distribute the required trust among the users of the service. It is not clear that either of these can be done at all.

5.1 Linking

Our first solution begins by observing that the sequence of clients requesting time-stamps and the hashes they submit cannot be known in advance. So if we include bits from the previous sequence of client requests in the signed certificate, then we know that the time-stamp occurred after these requests. But the requirement of including bits from previous documents in the certificate also can be used to solve the problem of constraining the time in the other direction, because the time-stamping company cannot issue later certificates unless it has the current request in hand.

We describe two variants of this linking scheme; the first one, slightly simpler, highlights our main idea, while the second one may be preferable in practice. In both variants, the TSS will make use of a collision-free hash function, to be denoted H. This is in addition to clients' use of hash functions in order to produce the hash value of any documents that they wish to have time-stamped.

To be specific, a time-stamping *request* consists of an l-bit string y (presumably the hash value of the document) and a client identification number ID. We use $\sigma(\cdot)$ to denote the signing procedure used by the TSS. The TSS issues signed, sequentially numbered time-stamp *certificates*. In response to the request (y_n, ID_n) from our client, the nth request in sequence, the TSS does two things:

1. The TSS sends our client the signed certificate $s = \sigma(C_n)$, where the certificate

$$C_n = (n, t_n, \mathrm{ID}_n, y_n; L_n)$$

   consists of the sequence number n, the time t_n, the client number ID_n and the hash value y_n from the request, and certain *linking information*, which comes from the previously issued certificate: $L_n = (t_{n-1}, \mathrm{ID}_{n-1}, y_{n-1}, H(L_{n-1}))$.

2. When the next request has been processed, the TSS sends our client the identification number ID_{n+1} for that next request.

Having received s and ID_{n+1} from the TSS, she checks that s is a valid signature of a good certificate, i.e. one that is of the correct form $(n, t, \mathrm{ID}_n, y_n; L_n)$, containing the correct time t.

If her time-stamped document x is later challenged, the challenger first checks that the time-stamp (s, ID_{n+1}) is of the correct form (with s being a signature of a certificate that indeed contains a hash of x). In order to make sure that our client has not colluded with the TSS, the challenger can call client ID_{n+1} and ask him to produce his time-stamp (s', ID_{n+2}). This includes a signature

$$s' = \sigma(n+1, t_{n+1}, \mathrm{ID}_{n+1}, y_{n+1}; L_{n+1})$$

of a certificate that contains in its linking information L_{n+1} a copy of her hash value y_n. This linking information is further authenticated by the inclusion of the image $H(L_n)$ of her linking information L_n. An especially suspicious challenger now can call up client ID_{n+2} and verify the next time-stamp in the sequence; this can continue for as long as the challenger wishes. Similarly, the challenger can also follow the chain of time-stamps backward, beginning with client ID_{n-1}.

Why does this constrain the TSS from producing bad time-stamps? First, observe that the use of the signature has the effect that the *only* way to fake a time-stamp is with the collaboration of the TSS. But the TSS cannot forward-date a document, because the certificate must contain bits from requests that immediately preceded the desired time, yet the TSS has not received them. The TSS

cannot feasibly back-date a document by preparing a fake time-stamp for an earlier time, because bits from the document in question must be embedded in certificates immediately following that earlier time, yet these certificates have already been issued. Furthermore, correctly embedding a new document into the already-existing stream of time-stamp certificates requires the computation of a collision for the hash function H.

Thus the only possible spoof is to prepare a fake chain of time-stamps, long enough to exhaust the most suspicious challenger that one anticipates.

In the scheme just outlined, clients must keep all their certificates. In order to relax this requirement, in the second variant of this scheme we link each request not just to the next request but to the next k requests. The TSS responds to the nth request as follows:

1. As above, the certificate C_n is of the form $C_n = (n, t_n, \mathrm{ID}_n, y_n; L_n)$, where now the linking information L_n is of the form

$$L_n = [(t_{n-k}, \mathrm{ID}_{n-k}, y_{n-k}, H(L_{n-k})), \ldots, (t_{n-1}, \mathrm{ID}_{n-1}, y_{n-1}, H(L_{n-1}))].$$

2. After the next k requests have been processed, the TSS sends our client the list $(\mathrm{ID}_{n+1}, \ldots, \mathrm{ID}_{n+k})$.

After checking that this client's time-stamp is of the correct form, a suspicious challenger can ask any one of the next k clients ID_{n+i} to produce his time-stamp. As above, his time-stamp includes a signature of a certificate that contains in its linking information L_{n+i} a copy of the relevant part of the challenged time-stamp certificate C_n, authenticated by the inclusion of the hash by H of the challenged client's linking information L_n. His time-stamp also includes client numbers $(\mathrm{ID}_{n+i+1}, \ldots, \mathrm{ID}_{n+i+k})$, of which the last i are new ones; the challenger can ask these clients for their time-stamps, and this can continue for as long as the challenger wishes.

In addition to easing the requirement that clients save all their certificates, this second variant also has the property that correctly embedding a new document into the already-existing stream of time-stamp certificates requires the computation of a simultaneously k-wise collision for the hash function H, instead of just a pairwise collision.

5.2 Distributed trust

For this scheme, we assume that there is a secure signature scheme so that each user can sign messages, and that a standard secure pseudorandom generator G is available to all users. A *pseudorandom generator* is an algorithm that stretches short input *seeds* to output sequences that are indistinguishable by any feasible algorithm from random sequences; in particular, they are unpredictable. Such generators were first studied by Blum and Micali [2] and by Yao [22]; Impagliazzo, Levin, and Luby have shown that they exist if there exist one-way functions [12].

Once again, we consider a hash value y that our client would like to time-stamp. She uses y as a seed for the pseudorandom generator, whose output can be interpreted in a standard way as a k-tuple of client identification numbers:

$$G(y) = (\mathrm{ID}_1, \mathrm{ID}_2, \ldots, \mathrm{ID}_k).$$

Our client sends her request (y, ID) to each of these clients. She receives in return from client ID_j a signed message $s_j = \sigma_j(t, \text{ID}, y)$ that includes the time t. Her time-stamp consists of $[(y, \text{ID}), (s_1, \ldots, s_k)]$. The k signatures s_j can easily be checked by our client or by a would-be challenger. No further communication is required in order to meet a later challenge.

Why should such a list of signatures constitute a believable time-stamp? The reason is that in these circumstances, the only way to produce a time-stamped document with an incorrect time is to use a hash value y so that $G(y)$ names k clients that are willing to cooperate in faking the time-stamp. If at any time there is at most a constant fraction ϵ of possibly dishonest clients, the expected number of seeds y that have to be tried before finding a k-tuple $G(y)$ containing only collaborators from among this fraction is ϵ^{-k}. Furthermore, since we have assumed that G is a secure pseudorandom generator, there is no faster way of finding such a convenient seed y than by choosing it at random. This ignores the adversary's further problem, in most real-world scenarios, of finding a plausible document that hashes to a convenient value y.

The parameter k should be chosen when designing the system so that this is an infeasible computation. Observe that even a highly pessimistic estimate of the percentage of the client population that is corruptible—ϵ could be 90%—does not entail a prohibitively large choice of k. In addition, the list of corruptible clients need not be fixed, as long their fraction of the population never exceeds ϵ.

This scheme need not use a centralized TSS at all. The only requirements are that it be possible to call up other clients at will and receive from them the required signatures, and that there be a public directory of clients so that it is possible to interpret the output of $G(y)$ in a standard way as a k-tuple of clients. A practical implementation of this method would require provisions in the protocol for clients that cannot be contacted at the time of the time-stamping request. For example, for suitable $k' < k$, the system might accept signed responses from any k' of the k clients named by $G(y)$ as a valid time-stamp for y (in which case a greater value for the parameter k would be needed in order to achieve the same low probability of finding a set of collaborators at random).

6 Remarks

6.1 Tradeoffs

There are a number of tradeoffs between the two schemes. The distributed-trust scheme has the advantage that all processing takes place when the request is made. In the linking scheme, on the other hand, the client has a short delay while she waits for the second part of her certificate; and meeting a later challenge may require further communication.

A related disadvantage of the linking scheme is that it depends on at least some clients storing their certificates.

The distributed-trust scheme makes a greater technological demand on the system: the ability to call up and demand a quick signed response at will.

The linking scheme only locates the time of a document between the times of the previous and the next requests, so it is best suited to a setting in which relatively many documents are submitted for time-stamping, compared to the scale at which the timing matters.

It is worth remarking that the time-constraining properties of the linking scheme do not depend

on the use of digital signatures.

6.2 Time constraints

We would like to point out that our scheme is not constrained to time-stamping both forward and backward in time. However, if any amount of time may pass between the creation of a document and when it is time-stamped, then no method can more than forward-constrain the time at which the document itself was created. Thus, in general, time-stamping should only be considered as evidence that a document has not been back-dated.

On the other hand, if the time-stamping event can be made part of the document creation event, then the constraint holds in both directions. For example, consider the sequence of phone conversations that pass through a given switch. In order to process the next call on this switch, one could require that linking information be provided from the previous call. Similarly, at the end of the call, linking information would be passed onto the next call. In this way, the document creation event (the phone call) includes a time-stamping event, and so the time of the phone call can be fixed in both directions. The same idea could be applied to sequential financial transactions, such as stock trades or currency exchanges, or any sequence of electronic interactions that take place over a given physical connection.

6.3 Theoretical considerations

Although we will not do it here, we suggest that a precise complexity-theoretic definition of the strongest possible level of time-stamping security could be given along the lines of the definitions given by Goldwasser and Micali [9], Goldwasser, Micali, and Rivest [10], and Galil, Haber, and Yung [8] for various cryptographic tasks. The time-stamping and the verification procedures would all depend on a *security parameter* p. A time-stamp scheme would be *polynomially secure* if the success probability of a polynomially bounded adversary who tries to manufacture a bogus time-stamp is smaller than any given polynomial inverse for sufficiently large p.

Under the assumption that there exist one-way claw-free permutations, we can prove our linking scheme to be polynomially secure. If we assume that there is always at most a constant fraction of corruptible clients, and assuming as well the existence of one-way functions (and therefore the existence of pseudorandom generators and of secure signature scheme), we can prove our distributed-trust scheme to be polynomially secure.

In §4.1 above, we mentioned the difference between "collision-free" and "universal one-way" hash functions. The existence of one-way functions is sufficient to give us universal one-way hash functions. However, in order to prove the security of our time-stamping schemes, we apparently need the stronger guarantee of the difficulty of producing hash collisions that is provided by the definition of collision-free hash functions. As far as is currently known, a stronger complexity assumption—namely, the existence of claw-free permutations—is needed in order to prove the existence of these functions. (See also [5] and [6] for further discussion of the theoretical properties of cryptographic hash functions.)

Universal one-way hash functions were the tool used in order to construct a secure signature scheme. Our apparent need for a stronger assumption suggests a difference, perhaps an essential

one, between signatures and time-stamps. It is in the signer's own interest to act correctly in following the instructions of a secure signature scheme (for example, in choosing a hash function at random from a certain set). For time-stamping, on the other hand, a dishonest user or a colluding TSS may find it convenient not to follow the standard instructions (for example, by choosing a hash function so that collisions are easy to find); the time-stamping scheme must be devised so that there is nothing to be gained from such misbehavior.

If it is possible, we would like to reduce the assumptions we require for secure time-stamping to the simple assumption that one-way functions exist. This is the minimum reasonable assumption for us, since all of complexity-based cryptography requires the existence of one-way functions [12, 13]

6.4 Practical considerations

As we move from the realm of complexity theory to that of practical cryptosystems, new questions arise. In one sense, time-stamping places a heavier demand on presumably one-way functions than would some other applications. For example, if an electronic funds transfer system relies on a one-way function for authentication, and that function is broken, then all of the transfers carried out before it was broken are still valid. For time-stamps, however, if the hash function is broken, then all of the time-stamps issued prior to that time are called into question.

A partial answer to this problem is provided by the observation that time-stamps can be renewed. Suppose we have two time-stamping implementations, and that there is reason to believe that the first implementation will soon be broken. Then certificates issued using the old implementation can be renewed using the new implementation. Consider a time-stamp certificate created using the old implementation that is time-stamped with the new implementation before the old one is broken. Prior to the old implementation's breaking, the only way to create a certificate was by legitimate means. Thus, by time-stamping the certificate itself with the new implementation, one has evidence not only that the document existed prior to the time of the new time-stamp, but that it existed at the time stated in the original certificate.

Another issue to consider is that producing hash collisions alone is not sufficient to break the time-stamping scheme. Rather, meaningful documents must be found which lead to collisions. Thus, by specifying the format of a document class, one can complicate the task of finding meaningful collisions. For example, the density of ASCII-only texts among all possible bit-strings of length N bytes is $(2^7/2^8)^N$, or $1/2^N$, simply because the high-order bit of each byte is always 0. Even worse, the density of acceptable English text can be bounded above by an estimate of the entropy of English as judged by native speakers [21]. This value is approximately 1 bit per ASCII character, giving a density of $(2^1/2^8)^N$, or $1/128^N$.

We leave it to future work to determine whether one can formalize the increased difficulty of computing collisions if valid documents are sparsely and perhaps randomly distributed in the input space. Similarly, the fact that a k-way linking scheme requires the would-be adversary to compute k-way collisions rather than collision pairs may be parlayed into relaxing the requirements for the hash function. It may also be worthwhile to explore when there exist hash functions for which there are *no* k-way collisions among strings in a suitably restricted subset of the input space; the security of such a system would no longer depend on a complexity assumption.

7 Applications

Using the theoretically best (cryptographically secure) hash functions, signature schemes, and pseudorandom generators, we have designed time-stamping schemes that possess theoretically desirable properties. However, we would like to emphasize the practical nature of our suggestion: because there are *practical* implementations of these cryptographic tools, both of our time-stamp schemes can be inexpensively implemented as described. Practical hash functions like Rivest's are quite fast, even running on low-end PC's [19].

What kinds of documents would benefit from secure digital time-stamping? For documents that establish the precedence of an invention or idea, time-stamping has a clear value. A particularly desirable feature of digital time-stamping is that it makes it possible to establish precedence of intellectual property without disclosing its contents. This could have a significant effect on copyright and patent law, and could be applied to everything from software to the secret formula for Coca-Cola.

But what about documents where the date is not as significant as simply whether or not the document has been tampered with? These documents can benefit from time-stamping, too, under the following circumstances. Suppose one can establish that either the necessary knowledge or the motivation to tamper with a document did not exist until long after the document's creation. For example, one can imagine a company that deals with large numbers of documents each day, some few of which are later found to be incriminating. If all the company's documents were routinely time-stamped at the time of their creation, then by the time it became apparent which documents were incriminating and how they needed to be modified, it would be too late to tamper with them. We will call such documents *tamper-unpredictable*. It seems clear that many business documents are tamper-unpredictable. Thus, if time-stamping were to be incorporated into the established order of business, the credibility of many documents could be enhanced.

A variation that may be particularly useful for business documents is to time-stamp a log of documents rather than each document individually. For example, each corporate document created in a day could be hashed, and the hash value added to the company's daily log of documents. Then, at the end of the business day, the log alone could be submitted for time-stamping. This would eliminate the expense of time-stamping each document individually, while still making it possible to detect tampering with each document; one could also determine whether any documents had been destroyed altogether.

Of course, digital time-stamping is not limited to text documents. Any string of bits can be time-stamped, including digital audio recordings, photographs, and full-motion videos. Most of these documents are tamper-unpredictable. Therefore, time-stamping can help to distinguish an original photograph from a retouched one, a problem that has received considerable attention of late in the popular press [1, 11]. It is in fact difficult to think of any other algorithmic "fix" that could add more credibility to photographs, videos, or audio recordings than time-stamping.

8 Summary

In this paper, we have shown that the growing use of text, audio and video documents in digital form and the ease with which such documents can be modified creates a new problem: how can one

certify when a document was created or last modified? Methods of certification, or time-stamping, must satisfy two criteria. First, they must time-stamp the actual bits of the document, making no assumptions about the physical medium on which the document is recorded. Second, the date and time of the time-stamp must not be forgeable.

We have proposed two solutions to this problem. Both involve the use of one-way hash functions, whose outputs are processed in lieu of the actual documents, and of digital signatures. The solutions differ only in the way that the date and time are made unforgeable. In the first, the hashes of documents submitted to a TSS are linked together, and certificates recording the linking of a given document are distributed to other clients both upstream and downstream from that document. In the second solution, several members of the client pool must time-stamp the hash. The members are chosen by means of a pseudorandom generator that uses the hash of the document itself as seed. This makes it infeasible to deliberately choose which clients should and should not time-stamp a given hash. The second method could be implemented without the need for a centralized TSS at all.

Finally, we have considered whether time-stamping could be extended to enhance the authenticity of documents for which the time of creation itself is not the critical issue. This is the case for a large class of documents which we call "tamper-unpredictable." We further conjecture that no purely algorithmic scheme can add any more credibility to a document than time-stamping provides.

Acknowledgements

We gratefully acknowledge helpful discussions with Donald Beaver, Shimon Even, George Furnas, Burt Kaliski, Ralph Merkle, Jeff Shrager, Peter Winkler, Yacov Yacobi, and Moti Yung.

References

[1] J. Alter. When photographs lie. *Newsweek*, pp. 44-45, July 30, 1990.

[2] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, Nov. 1984.

[3] G. Brassard and M. Yung. One-way group actions. In *Advances in Cryptology—Crypto '90*. Springer-Verlag, *LNCS*, to appear.

[4] I. Damgård. Collision-free hash functions and public-key signature schemes. In *Advances in Cryptology—Eurocrypt '87*, pp. 203-217. Springer-Verlag, *LNCS*, vol. 304, 1988.

[5] I. Damgård. A design principle for hash functions. In *Advances in Cryptology—Crypto '89* (ed. G. Brassard), pp. 416-427. Springer-Verlag, *LNCS*, vol. 435, 1990.

[6] A. DeSantis and M. Yung. On the design of provably secure cryptographic hash functions. In *Advances in Cryptology—Eurocrypt '90*. Springer-Verlag, *LNCS*, to appear.

[7] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Trans. on Inform. Theory*, vol. IT-22, Nov. 1976, pp. 644-654.

[8] Z. Galil, S. Haber, and M. Yung. Interactive public-key cryptosystems. *J. of Cryptology*, to appear.

[9] S. Goldwasser and S. Micali. Probabilistic encryption. *JCSS*, 28:270–299, April 1984.

[10] S. Goldwasser, S. Micali, and R. Rivest. A secure digital signature scheme. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[11] Andy Grundberg. Ask it no questions. The camera can lie. *The New York Times*, section 2, pp. 1, 29, August 12, 1990.

[12] R. Impagliazzo, L. Levin, and M. Luby. Pseudorandom generation from one-way functions. In *Proc. 21st STOC*, pp. 12-24. ACM, 1989.

[13] R. Impagliazzo and M. Luby. One-way functions are essential for complexity-based cryptography. In *Proc. 30th FOCS*, pp. 230-235. IEEE, 1989.

[14] H. M. Kanare. *Writing the laboratory notebook*, p. 117. American Chemical Society, 1985.

[15] R.C. Merkle. Secrecy, authentication, and public key systems. Ph.D. thesis, Stanford Univeristy, 1979.

[16] R.C. Merkle. One-way hash functions and DES. In *Advances in Cryptology—Crypto '89* (ed. G. Brassard), pp. 428-446. Springer-Verlag, *LNCS*, vol. 435, 1990.

[17] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proc. 21st STOC*, pp. 33-43. ACM, 1989.

[18] M.O. Rabin. Digitalized signatures. In *Foundations of Secure Computation* (ed. R.A. DeMillo et al.), pp. 155-168. Academic Press, 1978.

[19] R. Rivest. The MD4 message digest algorithm. In *Advances in Cryptology—Crypto '90*. Springer-Verlag, *LNCS*, to appear.

[20] J. Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proc. 22nd STOC*, pp. 387-394. ACM, 1990.

[21] C. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, vol. 30 pp. 50-64, 1951.

[22] A.C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd FOCS*, pp. 80-91. IEEE, 1982.

expending resources to add gold to circulation. In our case, it is CPU time and .electricity that is expended The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation .free The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth

Reclaiming Disk Space .7
Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the .block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The .interior hashes do not need to be stored

Block
Hash0 Hash1 Hash2 Hash3
Tx0 Tx1 Tx2 Tx3
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Block
Block Header (Block Hash)
Prev Hash
Nonce
Root Hash
Hash01
Hash23
Hash2 Hash3 Tx3
Transactions Hashed in a Merkle Tree After Pruning Tx0-2 from the Block

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be .kept in memory

Simplified Payment .8 Verification

It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle