Minimizing Bandwidth for Remote Access to Cryptographically Protected Audit Logs

John Kelsey and Bruce Schneier

Counterpane Internet Security, Inc.
101 E. Minnehaha Pkwy
Minneapolis, MN 55419
{*kelsey,schneier*}*@counterpane.com*

Abstract. Tamperproof audit logs are an essential tool for computer forensics. Building on the work in [SK98,SK99], we show how to build a tamperproof audit log where the amount of information exchange required to verify the entries in the audit log is greatly reduced. By making audit-log verification more efficient, this system is more suitable for implementation in low-bandwidth environments.

1 Introduction

Audit logs are an important tool for anyone trying to detect, understand, or assess the damage from an attack. Unfortunately, their forensic value makes audit logs an obvious target for attackers, who will often delete or change logfile entries that may reveal their presence. In [SK98,SK99], we developed a cryptographic mechanism for securing the contents of an audit log against alteration. That is, we created logs such that the logging machine itself is unable to read or undetectably alter previously-written log entries. Reading and verification of log entries is accomplished with the help of a trusted server: a remote machine on the network, or a remote machine to be dialed into, a tamper-resistant token (such as a smart card), or even a remote machine communicated with by way of mailing diskettes back and forth.

While no cryptographic mechanism can prevent an attacker from simply deleting all logs, our mechanism can ensure that:

1. The attacker cannot delete or alter the log file without certainty of detection.
2. The attacker cannot read encrypted log entries, and thus cannot see whether the logfile contains information that will reveal her presence.

Our audit log requires some interaction with a trusted server before writing audit logs, and also during any attempt to read encrypted audit log entries or verify the authenticity of the audit log. In our previous work, gaining access to these encrypted audit log entries required a great deal of bandwidth. In this paper, we describe a mechanism to enormously improve this, so that in many cases, access can be granted to large numbers of log entries, with only a few hundred bits of bandwidth.

The rest of the paper is arranged as follows: First, we give an overview of how our cryptographic audit logs are constructed. (This overview is mostly taken from [SK99].) Next, we describe our new method for making remote requests for access to encrypted log entries enormously more efficient. This method is developed in two rather different ways, and we expect these to be suitable for different kinds of applications. We also discuss potential problems with our improved scheme. We close with a brief summary and some open questions.

2 Secure Audit Logs and Cryptography

In this section, we briefly discuss our audit log mechanism. This should be considered only an overview; for a full understanding of the mechanism, see [SK99].

2.1 The Problem

Data kept on a machine is reliable only if an attacker has not compromised the machine. Once the machine has been compromised, an attacker can typically alter any data on the machine, without any real chance of detection, so long as the data isn't replicated somewhere else. It would be useful to keep audit logs on a machine with the property that they were available only for writing, not for deletion, updating, or reading. This would mean that a machine that was compromised at time t could not read, update, or delete any records written before time t.

Our basic solution to this problem involves having the logging machine (called $\mathcal{U}$ in the remainder of this paper) interact with a trusted server (called $\mathcal{T}$ in the remainder of this paper). A marginally-trusted verifier, $\mathcal{V}$, is able to both verify that the audit log hasn't been altered, and is able to get read access to encrypted records, by interacting with both $\mathcal{U}$ and $\mathcal{T}$.

Note that there are a number of limits on possible solutions to this problem. Once $\mathcal{U}$ is compromised, its log entries are inherently untrustworthy. No application of cryptography can alter this fact. The attacker cannot, however, delete, alter, or read old log entries without compromising the trusted machine, $\mathcal{T}$, or breaking one of the cryptographic mechanisms used for this application.

2.2 An Overview of Our Mechanism

The real goal of our audit log mechanism was to use cryptography to do access-control on a file, so that we can have a write-only file. To do this, we used a number of standard tools from cryptography:

1. An external trusted server, with which the logging machine must communicate in order to create a new logfile, and which may either directly verify a logfile's integrity, or interact with an verifier to do so.
2. Key-exchange mechanisms with perfect-forward secrecy, meaning that a compromise of either of the parties to the key-exchange mechanism after the exchange does not allow recovery of the key exchanged. An example of this is signed Diffie-Hellman key exchange.

3. Collision-resistant one-way hash functions, which guarantee that an attacker will neither be able to find messages that hash to the same value, nor to determine an input to the hash function that yields a given hash value as output. An example of this is the SHA1 hash function.
4. Hash chains, which allow us to keep a sort-of running hash of a file, which is both as secure as the underlying hash function with respect to collision-finding and preimage-finding attacks, and also efficient to update sequentially, one new record at a time.

The logfile is secured as follows:

1. When the log is created, a secret initial authentication key is established between $\mathcal{U}$ and $\mathcal{T}$.
2. Each time a log entry is written, the following steps are taken:
   (a) The log's authentication key is hashed, using a one-way hash function, immediately after a log entry is written. The new value of the authentication key overwrites and irretrieveably deletes the previous value.
   (b) Each log entry's encryption key is derived, using a one-way process, from that entry's authentication key. This makes it possible to give encryption keys for individual logs out to partially trusted users or entities (so that they can decrypt and read entries), without allowing those users or entities to make undetectable changes.
   (c) Each log entry contains an element in a hash chain that serves to authenticate the values of all previous log entries [HS91,SK97]. It is this value that is actually authenticated, which makes it possible to remotely verify all previous log entries by authenticating a single hash value.
   (d) Each log entry contains its own permission mask. This permission mask defines which log entries can be accessed by partially trusted users. Different partially trusted users can be given access to different kinds of entries. Because the encryption keys for each log entry are derived partly from the log entry type, lying about what permissions a given log entry has ensures that the partially trusted user simply never gets the right key.
3. When the log file is closed, a special kind of record is written, and the last authentication key is irretrieveably destroyed.

2.3 Log Entry Definitions and Construction Rules

All entries in the log file use the same format, and are constructed according to the following procedure:

1. D_j is the data to be entered in the jth log entry of ID_{log}. The specific data format of D is not specified in our scheme: it must merely be something that the reader of the log entries will unambiguously understand, and that can be distinguished in virtually all cases from random gibberish. (If we are dealing with raw binary data here, we may add some structure to the data to make detection of garbled information likely, though this is seldom going to be important.)

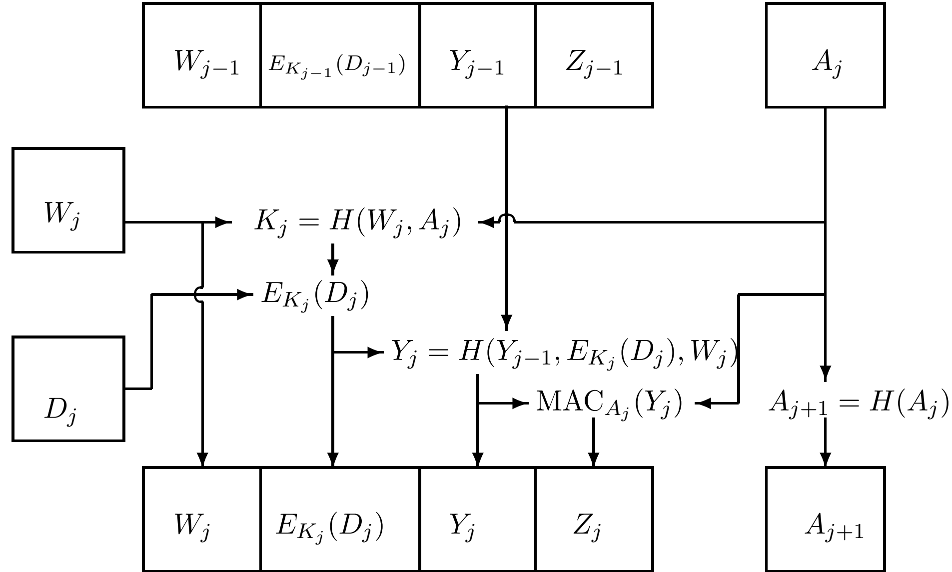

Fig. 1. Adding an entry to the log.

2. W_j is the log entry type of the jth log entry. This type serves as a permissions mask for $\mathcal{V}$; $\mathcal{T}$ will be allowed to control which log entry types any particular $\mathcal{V}$ will be allowed to access.
3. A_j is the authentication key for the jth entry in the log. This is the core secret that provides all of this scheme's security. Note that $\mathcal{U}$ must generate a new A_0 before starting the logfile; A_0 can be given to $\mathcal{U}$ by $\mathcal{T}$ at startup, or $\mathcal{U}$ can randomly generate it and then securely transmit it to $\mathcal{T}$.
4. $K_j = hash(\text{"Encryption Key"}, W_j, A_j)$. This is the key used to encrypt the jth entry in the log. Note that W_j is used in the key derivation to prevent the partially trusted verifier, $\mathcal{V}$, from getting decryption keys for log entry types to which he is not permitted access.
5. $Y_j = hash(Y_{j-1}, E_{K_j}(D_j), W_j)$. This is the hash chain that we maintain to allow partially trusted users, $\mathcal{V}$s, to verify parts of the log over a low-bandwidth connection with the trusted machine, $\mathcal{T}$. Y_j is based on $E_{K_j}(D_j)$ instead of D_j so that the chain can be verified without knowing the log entry. At startup, Y_{-1} is defined as a 20-byte block of binary zeros.[1]
6. $Z_j = MAC_{A_j}(Y_j)$.
7. $L_j = W_j, E_{K_j}(D_j), Y_j, Z_j$, where L_j is the jth log entry.
8. $A_{j+1} = hash(\text{"Increment Hash"}, A_j)$.

Note that when A_{j+1} and K_j are computed, the previous A_j and K_{j-1} values are irretrieveably destroyed; under normal operation there are no copies of these

[1] There is no security reason for this; it has to be initialized as something.

values kept on $\mathcal{U}$. Additionally, K_j is destroyed immediately after use in Step (4). (Naturally, an attacker will probably store A_j values after he takes control of $\mathcal{U}$.)

The above procedure defines how to write the jth entry into the log, given A_{j-1}, Y_{j-1}, and D_j. Figure 1 gives an illustration of this process.

If an attacker gains control of $\mathcal{U}$ at time t, he will have a list of valid log entries, $L_1, L_2, \ldots, L_t$, and the value A_{t+1}. He cannot compute A_{t-n} for any $n \leq t$, so he cannot read or falsify any previous entries. He can delete a block of entries (or the entire log file), but he cannot create new entries to replace the ones he deleted, regardless of whether he deleted a block of entries in the middle of the log or a block of entries at the end of the log. The next time $\mathcal{U}$ interacts with $\mathcal{T}$, $\mathcal{T}$ will realize that entries have been deleted from the log and that 1) $\mathcal{U}$ may have committed some invalid actions that have not been properly audited, and 2) $\mathcal{U}$ may have committed some valid actions whose audit record has been deleted.[2]

Even if the attacker does not delete entries, the audit system should include entries that point to a successful intrusion by the attacker. Again, since the attacker cannot read past entries, he will have no way of knowing if his intrusion was noted by the log system or not.

2.4 Startup: Creating the Logfile

In order to create a new logfile, $\mathcal{U}$ and $\mathcal{T}$ must somehow interact, so that the initial authentication key, A_0, can be agreed upon, transmitted, and committed to. The following requirements must be met:

1. A_0 must be sent to $\mathcal{T}$ in such a way that $\mathcal{U}$ would be unable to learn A_0 later by watching a recording of all messages sent between $\mathcal{T}$ and $\mathcal{U}$.
2. A_0 must be unambiguously committed to as an initial authentication key.
3. An initial start-of-log message must be written.

2.5 Closing the Logfile

Closing the logfile involves three operations: Writing the final-record message, D_f (the entry code is **NormalCloseMessage** and the data is a timestamp); irretrieveably deleting A_f and K_f; and (in some implementations) actually closing the file. Note that after the file has been properly closed, an attacker who has taken control of $\mathcal{U}$ cannot make any alteration to the logfile without detection. Nor can an attacker delete some entries (and possibly add others) and then create a valid close-file entry earlier in the log. Finally, the attacker cannot delete the whole log file, because of the earlier interaction between $\mathcal{U}$ and $\mathcal{T}$. Any of these changes will be detected when $\mathcal{T}$ sees the final logfile.

[2] If the attacker gains control of $\mathcal{U}$ before Step (8), he can learn A_t. In this case the tth log entry is not secured from deletion or manipulation.

2.6 Validating the Log

When $\mathcal{T}$ receives the complete and closed log, he can validate it using the hash chain and Z_f (since it already knows D_0). He can also derive all the encryption keys used, and thus read the whole audit log.

2.7 Verification, Verifiers, and Querying Entries

At times, a moderately trusted person or machine, called $\mathcal{V}$, may need to verify or read some of the logfile's records while they are still on $\mathcal{U}$. This is made possible if $\mathcal{T}$ has sent M_1 to $\mathcal{U}$ (see Section 3.3, and if $\mathcal{V}$ has a suitable channel available to and from $\mathcal{U}$. Note that this can occur before $\mathcal{T}$ has received a copy of the log from $\mathcal{U}$, and before $\mathcal{U}$ has closed the logfile.

1. $\mathcal{V}$ receives a copy of the audit log, $L_0, L_1, L_2, \ldots, L_f$, where f is the index value of the last record, from $\mathcal{U}$. Note that $\mathcal{U}$ does not have to send $\mathcal{V}$ a complete copy of the audit log, but it must send $\mathcal{V}$ all entries from L_0 through some L_f, including the entry with M_1.
2. $\mathcal{V}$ goes through the hash chain in the log entries (the Y_i values), verifying that each entry in the hash chain is correct.
3. $\mathcal{V}$ establishes a secure connection with $\mathcal{T}$.
4. $\mathcal{V}$ generates a list of all the log entries she wishes to read, from 0 to n. This list contains a representation of the log entry type and index of each entry to which the verifier is requesting access. (Typically, only some log entry types will be allowed, in accordance with $\mathcal{V}$'s permission mask.) This list is called $Q[0..n]$, where $Q_i = j, W_j$.
5. $\mathcal{V}$ forms and sends to $\mathcal{T}$:
   $M_2 = p, ID_{log}, f, Y_f, Z_f, Q[0..n]$.
6. $\mathcal{T}$ verifies that the log has been properly created on $\mathcal{U}$, and that $\mathcal{V}$ is authorized to work with the log. $\mathcal{T}$ knows A_0 so he can calculate A_f; this allows him to verify that $Z_f = MAC_{A_f}(Y_f)$. If there is a problem, $\mathcal{T}$ sends the proper error code to $\mathcal{V}$ and records that there is a problem with ID_{log} on $\mathcal{U}$ or a problem with $\mathcal{V}$.
7. If there are no problems, $\mathcal{T}$ forms a list, $R[1..n]$, of responses to the requests in Q. Each entry in Q gets a corresponding entry in R: either giving the verifier the decryption key for that record, or else giving it an error code describing the reason for the refusal of access. (Typically, this will be because the log entry type isn't allowed to be read by this verifier.) Note that $\mathcal{T}$ computes these keys based on the log entry type codes given. If $\mathcal{V}$ provides incorrect codes to $\mathcal{T}$, the keys will be incorrect and $\mathcal{V}$ will be unable to decrypt the log entries. Additionally, $\mathcal{V}$ will not be able to derive the right key from the key he has been given.
8. $\mathcal{T}$ forms and sends to $\mathcal{V}$:
   $M_3 = p, R[0..n]$.
9. $\mathcal{V}$ is now able to decrypt and read, but not to change, the log entries whose keys were sent in M_4. $\mathcal{V}$ is also convinced that the log entries are authentic, since a correct MAC on any hash-chain value is essentially a MAC on all previous entries as well.

10. $\mathcal{V}$ deletes the key it established with $\mathcal{T}$ in Step (3). This guarantees that an attacker cannot read $\mathcal{U}$'s logfile if $\mathcal{V}$ is compromised later.

Of course, if $\mathcal{V}$ is compromised at the start of this protocol with $\mathcal{T}$, it will be able to read $\mathcal{U}$'s logfile. Presumably, $\mathcal{V}$ will have its own authenticated logfiles and will be regularly audited to reduce the likelihood of this problem occuring.

Note that the above describes the audit log as it was defined in [SK99], and is mostly taken from that paper. Below, we will discuss a way to radically decrease the bandwidth needed to grant $\mathcal{V}$ read access to a sequence of records in the log.

3 Minimizing Bandwidth for Offline Access

The basic audit log scheme, as described above, allows a trusted machine, which interacted with the logging machine before the first log entry was written, to read and verify logs made on that machine later. However, some applications require authentication or decryption of logs on-site, without shipping the whole logfile to the remote trusted machine. For this reason, we developed ways to allow the trusted machine to grant to a marginally trusted verifier the following powers:

1. The ability to verify the authenticity of the log entries, requiring one hash output (20 bytes for SHA1) to and from the trusted machine.
2. The ability to read encrypted log entries, requiring a variable-length request to the trusted machine specifying the records to be read, and one encryption key (14 bytes for two-key triple-DES) per record.

While this is reasonable for access to small numbers of records, the bandwidth requirements quickly become unacceptable when we need to grant a verifier access to large numbers of encrypted entries. This is especially true when access to the trusted machine, $\mathcal{T}$, is over a low-bandwidth connection.

In this section, we develop a new way to grant access to encrypted records. This requires a change in the way the entry encryption keys (K_j, above) are computed, and it requires that the initial exchanged key, A_0, be used to derive a few additional keys before being used. We get the following results from these changes:

1. We can design the logfile so that granting access to some standard-sized block of records, e.g., records 0-99, 100-199, etc., requires only one key (about 100 bits) to be sent down to the verifier.
2. We can design the logfile so that granting access to arbitrary subsets of records, by time or record number, can be done with a minimal amount of bandwidth, e.g., on the order of $100 \times log(N)$ bits of key material for N records.

3.1 The General Idea

The basic idea behind our improved mechanism is to allow the verifier, $\mathcal{V}$, to compute many of his own keys for records to which he is granted access. Instead of giving him the keys for records 100-199, we would like to give him the seed material used by the logging machine, $\mathcal{U}$, to compute those records' encryption keys originally.

Clearly, we cannot simply use a single hash chain, as is done for the authentication key. If record encryption key i were derived as $K_i = hash(K_{i-1})$, we could give $\mathcal{V}$ the ability to compute many keys for himself, but there would be no limit. If we gave him K_{155}, he could compute all remaining encryption keys used from then on.

To avoid this, we must use more than one hash chain. In fact, we build a sort of a bank of "counters" with our different hash chains. For a simple example, consider a system with two hash chains:

$$X_i = \begin{cases} hash(\text{“Log Key”}, X_{i-1}), & \text{if } i \bmod 10 = 0 \\ X_{i-1}, & \text{otherwise} \end{cases}$$
$$K_i = \begin{cases} hash(\text{“Record Key”}, X_i, K_{i-1}), & \text{if } i \bmod 10 = 0 \\ hash(\text{“Record Key”}, K_{i-1}), & \text{otherwise} \end{cases}$$

With this set of keys and rules, we can give $\mathcal{V} K_0$, and he can compute $K_{1..9}$. We can give him K_3 and he can compute $K_{4..9}$. However, he can't go past record 10, since that depends upon the log key. We are thus able to give $\mathcal{V}$the power to compute the keys for blocks of records. Note that it is not possible, using this scheme, to give $\mathcal{V}$access to only record 11. Once he has K_{11}, he can compute $K_{12..19}$.

3.2 Easy Access to Ranges of Records

We can use this basic idea to do something pretty useful. Consider a system in which verifiers might be granted access to audit log records in hour-long blocks, e.g., 1:00:00-1:59:59, 2:00:00-2:59:59, etc. We have two keys of interest:

$$X_i = \begin{cases} hash(\text{“Log Key”}, X_{i-1}), & \text{if hour mark has passed} \\ X_{i-1}, & \text{otherwise} \end{cases}$$
$$K_i = \begin{cases} hash(\text{“Record Key”}, X_i, K_{i-1}), & \text{if hour mark has passed} \\ hash(\text{“Record Key”}, K_{i-1}), & \text{otherwise} \end{cases}$$

That is, we compute record keys as independent hash chains for one-hour blocks. Once an hour, a value from another hash chain is injected into the computation. A verifier, $\mathcal{V}$, can be given access to all records in one of these standard one-hour block by giving him the first record key in that hour block.

Consider an attacker who takes control of the logging machine and tries to read old log entries. As long as old K_i values are deleted and never stored, the attacker cannot learn any, because she would have to invert $K_i = hash(K_{i-1})$.

We can now tell a verifier the key K_i for the beginning of some hour. The verifier is then able to compute K_i values for all records in the log during that hour. For many applications, this is the optimal way to grant remote access to log entries. At the end of the hour, a new X_i is generated, K_i is made dependent on it, and thus, the verifier will no longer be able to compute any future K_i values. We can give the verifier access to any number of hour blocks, without his ever being able to compute a key we haven't allowed him to have. On the other hand, access can only be given for sequences of records reaching 'till the end of the hour.

Advantages and Limitations The most important limitation to this scheme is that we are able to grant access only in these one-hour blocks. We can neither limit access to the records in one 15-minute period, nor efficiently grant access to a whole week's records. (To grant access to a week's records, we would have to send $\mathcal{V}$ 168 different keys.) Another limitation is the inability to use the permission masks to restrict access to the keys. We will discuss ways to get around this limitation below.

The advantage of this scheme is its simplicity. If we can choose a standard kind of block of log entries to grant access to, whether that's all log entries in each hour, or in each week, or each block of 100 or 10,000 log entries, then we can grant access very efficiently from the trusted machine to the verifier. For many systems, this is exactly the kind of access that's needed.

3.3 Tree-Based Access to Records

This basic idea can be extended in a couple of ways, to give us extreme flexibility in the ranges of records to be revealed to some verifier. The cost of this is added complexity, and also added communications overhead when all we really want to do is to grant access to some standard-sized block of records.

Adding this flexibility gives us two requirements:

1. Each record key must be computed using some information besides the previous record key. Otherwise, knowledge to one record key will generally give the ability to compute others.
2. It must be possible to grant access to long stretches of log entries with only a few keys. Otherwise, the system won't be flexible enough to handle long stretches of log entries efficiently.

Here is an example of a set of keys and update rules that fits these requirements:

$$K[1\ 000]_j = \begin{cases} hash(\text{``1000''}, K[1\ 000]_{j-1}), & \text{if } j \bmod 1\ 000 = 0 \\ K[1\ 000]_{j-1}, & \text{otherwise} \end{cases}$$

$$K[100]_j = \begin{cases} hash(\text{``100''}, K[1\ 000]_j, K[100]_{j-1}), & \text{if } j \bmod 10 = 0 \\ K[100]_{j-1}, & \text{otherwise} \end{cases}$$

$$K[10]_j = \begin{cases} hash(\text{"10"}, K[100]_j, K[10]_{j-1}), & \text{if } j \bmod 10 = 0 \\ K[10]_{j-1}, & \text{otherwise} \end{cases}$$
$$K[1]_j = hash(\text{"1"}, K[10]_j, K[1]_{j-1})$$

In words, each time we're computing the next key, we always use a key from the next level up.

Now, consider granting access to records 000-225. This would amount to:

1. Granting access to records 000-099 by giving $\mathcal{V}$ keys $K[100]_0, K[10]_0, K[1]_0$. This would allow $\mathcal{V}$ to compute $K[1]_1..K[1]_{99}$. $\mathcal{V}$ could not compute $K[1]_{100}$, however, because of his lack of knowledge of $K[100]_{100}$.
2. Granting access to records 100-199 in the same way, by giving $\mathcal{V}$ key $K[100]_{100}$. Now, $\mathcal{V}$ could compute up to $K[1]_{199}$, but his lack of knowledge of $K[100]_{200}$ would prevent him from computing $K[1]_{200}$.
3. Granting access to records 200-209 by giving $\mathcal{V}$ key $K[10]_{200}$. Note that we cannot give $\mathcal{V}$ key $K[100]_{200}$, or he would be able to compute keys for records 200-299.
4. Granting access to records 210-219 by giving $\mathcal{V}$ key $K[10]_{210}$.
5. Granting access to records 220-225 by giving $\mathcal{V}$ keys $K[1]_{220..225}$.

In this example, we would give $\mathcal{V}$ a total of 12 keys to grant him access to 226 records.

Consider another example. We wish to give $\mathcal{V}$ access to records 121-881. We do this by:

1. Giving $\mathcal{V}$ access to records 121-129 by giving him $K[1]_{121,122,...,129}$.
2. Giving $\mathcal{V}$ access to records 130-199 by giving him $K[10]_{130,140,...,190}$. He uses $K[1]_{129}$ and $K[10]_{130}$ to compute records $K[1]_{130..139}$, uses $K[10]_{140}$ to compute records $K[1]_{140..149}$, and so on.
3. Giving $\mathcal{V}$ access to records 200-799 by giving him $K[100]_{200,300,400,...,700}$. He uses records $K[1]_{199}, K_[10]_{199}, K[100]_{200}$ to compute $K[1]_{200..299}, K[10]_{200..299}$, and so on.
4. Giving $\mathcal{V}$ access to records 800-879 by giving him $K[10]_{800,810,...,870}$.
5. Giving $\mathcal{V}$ access to records 880-881 by giving him $K[1]_{880,881}$.

Note that in giving $\mathcal{V}$ access to records 121-199, we must not simply give him $K[100]_{121}, K[10]_{121}, K[1]_{121}$. If we did that, and later gave him access to records 101-109 by giving him keys $K[10]_{101}, K[1]_{101}$, he would be able to compute the keys to records 110-120, without our access. By not giving him $K[100]_{121} = K[100]_{100}$, we avoid this problem.

The underlying idea is that we can give $\mathcal{V}$ the ability to compute decryption records for himself, but only for a limited range of records. By giving him $K[100]_0, K[10]_0, K[1]_0$, we give him the ability to compute keys from record 0 to record 99. At that point, $K[100]$ changes, and since $\mathcal{V}$ doesn't know what $K[1\,000]$ is, he cannot go any further.

Our examples above have been in terms of base ten denominations of keys, but the same idea works for any base. That is, we can as easily have $K[1, 2, 4, 8, 16, ..., 2^{24}]$

as $K[1, 10, 100, 1000, ..., 1000000]$. Although we have not done extensive experimentation, the differences between denominating keys in base 2 and base 10 in terms of number of keys transmitted appear to be fairly small.

There is an important limitation to the above set of examples. We must never give $\mathcal{V}$ access to the top-level key, since he could then compute all top-level key values after that. This means that our system would become inefficient if we had to deal with ranges of several thousand records in a request in the above examples, since we could never grant access to the **thousands** key without potentially giving away all keys in the remainder of the log file. For example, consider granting access to records 42000 through 48000. $\mathcal{V}$ would have to be given $K[100]_{42000,42100,42200,...,47\,900}$. This would require 60 keys to be transmitted.

In general, then, the largest denomination of key should be chosen to be larger than the largest range of records to which we ever expect to grant access. If we expect to never grant access to more than 1 000 records at a time, then the above set of keys is acceptable. Otherwise, we should add a **ten thousands** key, a **hundred thousands** key, etc., until the largest denomination of key is larger than the longest sequence of records we expect to give to $\mathcal{V}$.

3.4 Time-Based Access

A similar idea can be done based on increments of time. Instead of a key for ones, tens, hundreds, and thousands, we can have a key for records, hours, days, and weeks. We would have the same kind of update rules, e.g.:

$$K[weeks]_j = \begin{cases} hash(\text{“weeks”}, K[weeks]_{j-1}), & \text{if a new week has started} \\ K[weeks]_{j-1}, & \text{otherwise} \end{cases}$$

$$K[days]_j = \begin{cases} hash(\text{“days”}, K[weeks]_j, K[days]_{j-1}), & \text{if a new day has started} \\ K[days]_{j-1}, & \text{otherwise} \end{cases}$$

$$K[hours]_j = \begin{cases} hash(\text{“hours”}, K[days]_j, K[hours]_{j-1}), & \text{if a new hour has started} \\ K[hours]_{j-1}, & \text{otherwise} \end{cases}$$

$$K[records]_j = hash(\text{“records”}, K[hours]_j, K[records]_{j-1})$$

This demonstrates a bit more of the flexibility of this approach. We can now grant very efficient access to one hour blocks, as before (by sending just two keys, $K[hours]_j$ and $K[records]_j$), and we may also efficiently grant access to blocks of log entries for up to about a day in length.

For example, consider granting $\mathcal{V}$ access to all records from 11:00 PM on Dec 11 through 10:00 PM Dec 15. We would have to give $\mathcal{V}$:

1. $K[records]_{Dec11,11:00PM,0}, K[hours]_{11PM}$, which will allow him to compute all keys through 12:00AM on Dec 12.
2. $K[days]_{Dec12,Dec13,Dec14}$, which will now allow him to compute all keys on those days.
3. $K[hours]_{12AM,1AM,2AM,...,10PM}$ from Dec 15, which will allow him to compute keys for all those hours on Dec 15.

We thus must send 27 keys to give access to all log entries between 11:00 PM Dec 11 and 10:00 PM Dec 15.

3.5 Permission Masks

In our previous scheme for granting access to encrypted records of the logfile, we made use of permission masks to restrict access to certain kinds of records. That is, if $\mathcal{V}$requested the key for record 239, claiming that it was a kind of record he had access to, he would be given a key that would work only if the record was the kind he had claimed. This is important, because this remote access is happening with $\mathcal{V}$ in possession of an encrypted log, and $\mathcal{U}$ not able to see the log. There is nothing to keep $\mathcal{V}$ from lying about what kind of permission mask a given record has.

Our improved scheme requires the ability for $\mathcal{V}$ to compute keys for himself. This means we cannot use our permission mask mechanism any longer. Instead, we must use a much less efficient mechanism. We will have to maintain a whole set of keys for deriving record keys, as described above, for each different permission mask. This means that instead of potentially having millions of different permission masks, we will likely end up with two or three.

4 Conclusions and Open Questions

In this paper, we have given an overview of our secure audit logging mechanism, and presented an extension that enormously increases its usefulness in environments where the trusted machine can be contacted only over a low-bandwidth channel.

Some important questions remain open. Among them:

1. What is the most efficient way to denominate the keys in our scheme, given various expected patterns of requested records?
2. Is there a way to improve our handling of permission masks for various kinds of log entries?
3. What other variations to our logging scheme will make it more valuable and useful in network security and intrusion detection environments?

References

[HS91] S. Haber and W.S. Stornetta, "How to Time Stamp a Digital Document," *Advances in Cryptology — Crypto '90*, Springer-Verlag, 1991, pp. 437–455.

[SK97] B. Schneier and J. Kelsey, "Automatic Event-Stream Notarization Using Digital Signatures," *Security Protocols, International Workshop, Cambridge, United Kingdom, April 1996 Proceedings*, Springer-Verlag, 1997, pp. 155–169.

[SK98] B. Schneier and J. Kelsey, "Cryptographic Support for Secure Logs on Untrusted Machines," *The Seventh USENIX Security Symposium Proceedings*, USENIX Press, Jan 1998, pp. 53–62.

[SK99] B. Schneier and J. Kelsey, "Tamperproof Audit Logs as a Forensics Tool for Intrusion Detection Systems," *Computer Networks and ISDN Systems*, 1999, to appear.