Efficient Record-Level Keyless Signatures for Audit Logs

Ahto Buldas[1], Ahto Truu[1], Risto Laanoja[1], and Rainer Gerhards[2]

[1] Guardtime AS, Tallinn, Estonia
{ahto.buldas,ahto.truu,risto.laanoja}@guardtime.com
[2] Adiscon GmbH, Großrinderfeld, Germany
rgerhards@adiscon.com

Abstract. We propose a log signing scheme that enables (a) verification of the integrity of the whole log, and (b) presentation of any record, along with a compact proof that the record has not been altered since the log was signed, without leaking any information about the contents of other records in the log. We give a formal proof of the security of the proposed scheme, discuss practical considerations, and provide an implementation case study.

Keywords: applied security, secure logging, keyless signatures, cryptographic time-stamps, `syslog`, `rsyslog`.

1 Introduction

Increasingly, logs from various information systems are used as evidence. With that trend, also the requirements on maintenance and presentation of the log data are growing. Availability is clearly the most important property. If the logs are prematurely erased or can't be located when needed, any other qualities of the log data don't really matter much. However, ensuring availability is outside of the scope of the current discussion.

Integrity and authenticity—confidence that the information in the log has not been tampered with or the log replaced with another one altogether—are also quite obvious requirements, especially if the log data is to be used for dispute resolution or admitted as evidence in legal proceedings. Digital signing and time-stamping are standard solutions for proving authenticity and integrity of data.

As information systems log all their activities in a sequential manner, often the details of the transactions involved in the dispute are interspersed with other information in a log. To protect the confidentiality of the unrelated events, it is then desirable to be able to extract only some records from the signed log and still prove their integrity.

An example of such a case is a dispute between a bank and a customer about a particular transaction. On one hand, the bank can't just present the whole log as evidence, as the log contains also information about transactions of other customers. On the other hand, the customer involved in the dispute (or more likely a technical expert working on their behalf) should have a chance to verify the integrity of the records relevant to the dispute.[3] Similar concerns also arise in the context of multi-tenant cloud environments.

In the light of the above, an ideal log signing scheme should have the following properties:

- The integrity of the whole log can be verified by the owner of the log: no records can be added, removed or altered undetectably.
- The integrity of any record can be proven to a third party without leaking any information about the contents of any other records in the log.
- The signing process is efficient in both time and space. (Ideally, there is a small constant per-record processing overhead and a small constant per-log storage overhead.)
- The extraction process is efficient in both time and space. (Ideally, a small constant-sized proof of integrity can be extracted for any record in time sub-linear in the size of the log.)
- The verification process is efficient in time. (Ideally, it should be running in time linear in the size of the data to be verified—whether verifying the whole log or a single record.)

[3] The authors have in fact been asked for such a feature in private communication with some European financial institutions.

1.1 Related Work

Schneier and Kelsey [13] proposed a log protection scheme that encrypts the records using one-time keys and links them using cryptographic hash functions. The scheme allows both for verification of the integrity of the whole log and for selective disclosure of the one-time encryption keys. However, it needs a third party trusted by both the logger and the verifier and requires active participation of this trusted party in both phases of the protocol.

Holt [7] replaced the symmetric cryptographic primitives used in the protocol by Schneier and Kelsey with asymmetric ones and thus enabled verification without the trusted party. However, Holt's scheme requires public-key signatures on individual records, which adds high computational and storage overhead to the logging process. Also, the size of the information required to prove the integrity of one record is at least proportional to the square root of the distance of the record from the beginning of the log. Other proposed amendments [14, 1] of the original protocol have similar weaknesses.

Kelsey *et al* [10] proposed a log signing scheme where records are signed in blocks, by first computing a hash value of each record in a block and then signing the sequence of hash values. This enables efficient verification of the integrity of the whole block, significantly reduces the overhead compared to having a signature per record, and also removes the need to ship the whole log block when a single record is needed as evidence. But still the size of the proof of a record is linear in the size of the block. Also, other records in the same block are not protected from the informed brute-force attack discussed in Sec. 2.1.

Ma and Tsudik [11] utilised the authentication technique they called *Forward-Secure Sequential Aggregate* to construct a logging protocol which provides *forward-secure stream integrity*, retaining the provable security of the underlying primitives. They also proposed a possibility to store individual signatures in the log file to gain better signature granularity, at the expense of storage efficiency. Based on the performance comparison table provided in [11], where best-case signer computation cost is 5.55 ms per log entry (albeit on slightly weaker hardware compared to our experiment), we can estimate that our scheme is two to three orders of magnitude faster.

2 Data Model

In this section we present the design of a signing scheme that will allow us to achieve almost all of the goals (there will be some trade-offs on the efficiency goals, but no compromises on the security goals).

A computational process producing a log may, in principle, run indefinitely and thus the log as an abstract entity may not have a well-defined beginning and end. In the following, we model the log as an ordered sequence of blocks, where each block in turn is an ordered sequence of a finite number of records. Many practical logging systems work this way, for example in the case of `syslog` output being sent to a log file that is periodically rotated.

The most straightforward strategy—signing each record individually—would, of course, have very high overhead in both processing and storage, as signing is quite expensive operation and the size of a signature may easily exceed the size of a typical log record. More importantly, it would also fail to fully ensure the integrity of the log as a whole—deletion of a record along with its signature would not be detected. The next simplest possible strategy—signing each log block as a unit—would satisfy all the requirements related to processing of the whole block, but would make it impossible to prove the integrity of individual records without exposing everything else in the block.

An improvement over both of the above naive strategies would be to compute a hash value of each record in a log block and then sign the sequence of hash values instead of the records themselves (as proposed by Kelsey *et al* in [10]). This would ensure the integrity of the whole log block, significantly reduce the overhead compared to signing each record separately and also remove the need to ship the whole log block when a single record is needed as evidence. But still the size of the proof of a record would be linear in the size of the block (and the latter could easily run into multiple millions of records for a busy system).

2.1 Merkle Trees with Blinding Masks

To further reduce the size of the evidence for a single record, the records can instead be aggregated using a data structure known as Merkle tree—a binary tree whose leaves are the hash values of the records and each non-leaf node is the hash value of the concatenation of the values in its child nodes. The hash value in the root node of the tree can

then be signed and for each leaf node a compact (logarithmic in the number of leaves) proof extracted showing that the hash value in the leaf participated in the computation that led to the signed root hash value. [12]

There are two complications left to be dealt with. The first one is that the security of such an aggregation scheme against retroactive fabrication of hash chains can in general be proven only if some restrictions are placed on the shapes of the hash chains allowed as participation proofs. Fortunately, not much is needed: appending the height of the sub-tree to the concatenated hash values from the child nodes before hashing is sufficient. This limits the length of the hash chains accepted during verification and allows for the security of the scheme to be formally proven. [4]

The second complication is that the hash chain extracted from the Merkle tree for one node contains hash values of other nodes. A strong hash function can't be directly reversed to learn the input value from which the hash value in the chain was created. However, a typical log record may contain insufficient entropy to make that argument—an attacker who knows the pattern of the input could exhaustively test all possible variants to find the one that yields the hash value actually in the chain and thus learn the contents of the record. To prevent this kind of informed brute-force attack, a blinding mask with sufficient entropy can be added to each record before aggregating the hash values.

Generating cryptographically secure random values is expensive. Additionally, when an independent random value would be used as a blinding mask for each record, all these values would have to be stored for later verification. It is therefore much more efficient to derive all the blinding masks from a single random seed, as in the data structure shown on Fig. 1, where each node with incoming arrows contains the hash value of the concatenation of the contents of the respective source nodes.

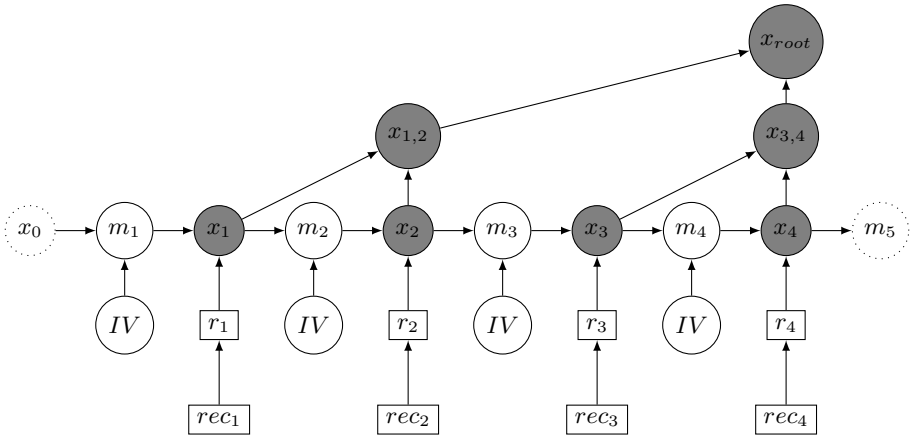

Fig. 1. Log signing using a Merkle tree with interlinks and blinding masks: rec_i are the log records; r_i are the hash values of the records; IV is the random seed; m_i are the blinding masks; x_i are leaves and $x_{a,b}$ are internal nodes of the Merkle tree; x_{root} is the value to be signed.

3 Security Proof

In this section we show that the integrity proofs for any set of records do not leak any information about the contents of any other records. The security of the scheme against modification of the log data has already been shown in [4].

We give the proof under the PRF assumption. Informally, the PRF assumption means that a 2-to-1 hash function $h\colon \{0,1\}^{2n} \to \{0,1\}^n$ can be assumed to behave like a random function $\Omega\colon \{0,1\}^n \to \{0,1\}^n$ when the first half of the input is a randomly choosen secret value $r \leftarrow \{0,1\}^n$.

Definition 1 (Pseudo-Random Function Family, PRF). *By an S-secure* pseudorandom function family *we mean an efficiently computable two-argument function h, such that if the first argument r is randomly chosen then the one-argument function $h(r,\cdot)$ (given to the distinguisher as a black box without direct access to r) is S-indistinguishable*

from the true random oracle Ω of the same type, i.e. for any t-time distinguisher D:

$$\mathsf{Adv}(D) = \left| \Pr\left[1 \leftarrow D^{h(r,\cdot)}\right] - \Pr\left[1 \leftarrow D^{\Omega(\cdot)}\right] \right| \leq \frac{t}{S},$$

where $r \leftarrow \{0,1\}^n$, $h\colon \{0,1\}^n \times \{0,1\}^p \to \{0,1\}^m$ and $\Omega\colon \{0,1\}^p \to \{0,1\}^m$.

Note that the PRF assumption is very natural when h is a 2-to-1 hash function, considering the design principles of hash functions, especially of those constructed from block ciphers.

Definition 2 (Indistinguishability under Chosen-Plaintext Attack, IND-CPA). *The log-signing scheme is said to be S-secure IND-CPA content concealing, if any t-time adversary $A = (A_1, A_2)$ has success probability $\delta \leq \frac{t}{S}$ in the following attack scenario (Fig. 2, left):*

1. *The first stage A_1 of the adversary chooses the position i and a list of records $rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell$, as well as two test records rec_i^0, rec_i^1 and an advice string a.*
2. *The environment picks randomly $x_0 \leftarrow 0^n$, $IV \leftarrow \{0,1\}^n$ and $b \leftarrow \{0,1\}$, assigns $rec_i \leftarrow rec_i^b$ and for every $j \in \{1, \ldots, \ell\}$ computes: $m_j \leftarrow h(IV, x_{j-1})$, $r_j \leftarrow \mathsf{H}(rec_j)$, and $x_j \leftarrow h(m_j, r_j)$.*
3. *The second stage A_2 of the adversary, given as input the advice string a and the lists of hash values $x_1, \ldots, x_\ell$, and $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell$, tries to guess the value of b by outputting the guessed value $\hat{b}$.*

The advantage of A is defined as $\delta = 2 \left| \Pr\left[\hat{b} = b\right] - \frac{1}{2} \right|$.

Theorem 1. *If $h(IV, \cdot)$ is an S-secure pseudorandom function family, then the log-signing scheme is $\frac{S}{4}$-secure IND-CPA content concealing.*

Proof. Let A be a t-time adversary with success δ. We define three games Game_0, Game_1, and Game_2, where Game_0 is the original attack game, Game_2 is a simulator in which the input of A_2 does not depend on b and hence $\Pr\left[\hat{b} = b\right] = \frac{1}{2}$ in this game (Fig. 2), and Game_1 is an intermediate game, where A tries to break the scheme with independent random masks (Fig. 3), i.e. the mask generation steps $m_j \leftarrow h(IV, x_{j-1})$ are replaced with independent uniform random choices $m_j \leftarrow \{0,1\}^n$. However, as m_j is a function of x_{j-1}, we will use the same random number for m_j and m_k in case $x_{j-1} = x_{k-1}$. This allows us to view these random numbers m_j as outputs of a random oracle Ω and perfect simulation is possible.

Let δ_i ($i = 0, 1, 2$) denote the probability that the adversary correctly guessed the value of b in the i-th game, i.e. that $\hat{b} = b$ in the game Game_i. Hence, $\delta_2 = \frac{1}{2}$ and $\delta = 2|\delta_0 - \delta_2|$.

Game_0 or the original attack game:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$

2a. $x_0 \leftarrow 0^n$, $IV \leftarrow \{0,1\}^n$, $b \leftarrow \{0,1\}$

2b. $rec_i \leftarrow rec_i^b$

2c. $\forall j \in \{1, \ldots, \ell\}$:

$$m_j \leftarrow h(IV, x_{j-1})$$
$$r_j \leftarrow \mathsf{H}(rec_j)$$
$$x_j \leftarrow h(m_j, r_j)$$

3. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
4. **if** $\hat{b} = b$ **then** output 1 **else** output 0

Game_2 or the Simulator $\mathcal{S}$:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$

2a. $x_0 \leftarrow 0^n$, $b \leftarrow \{0,1\}$

2b. $rec_i \leftarrow rec_i^b$

2c. $\forall j \in \{1, \ldots, \ell\}$:

if $x_{j-1} = x_{k-1}$ for some $k < j$ **then** $m_j \leftarrow m_k$
else $m_j \leftarrow \{0,1\}^n$
$r_j \leftarrow \mathsf{H}(rec_j)$
if $j = i$ **then** $x_j \leftarrow \{0,1\}^n$ **else** $x_j \leftarrow h(m_j, r_j)$

3. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$
4. **if** $\hat{b} = b$ **then** output 1 **else** output 0

Fig. 2. The original attack game and the simulator.

Game_1:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$

2a. $x_0 \leftarrow 0^n, b \leftarrow \{0,1\}$

2b. $rec_i \leftarrow rec_i^b$

2c. $\forall j \in \{1, \ldots, \ell\}$:

if $x_{j-1} = x_{k-1}$ for some $k < j$ **then** $m_j \leftarrow m_k$
else $m_j \leftarrow \{0,1\}^n$
$r_j \leftarrow \mathsf{H}(rec_j)$
$x_j \leftarrow h(m_j, r_j)$

3. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$

4. **if** $\hat{b} = b$ **then** output 1 **else** output 0

Fig. 3. The intermediate game Game_1.

We will now show that the games are negligibly close in terms of adversary's advantage, and hence the advarsary's success cannot be considerably higher in the original attacking game Game_0 compared to the advantage of the simulator $\mathcal{S}$. To show that Game_0 is close to Game_1, we define a distinguisher D_{01}^{Φ} (with running time $t_1 \approx t$) between $\Phi = \Omega$ and $\Phi = h(r, \cdot)$ with success at least $|\delta_0 - \delta_1|$. To show that Game_1 is close to Game_2, we define a distinguisher D_{12}^{Φ} (with running time $t_2 \approx t$) between $\Phi = \Omega$ and $\Phi = h(r, \cdot)$ with success at least $|\delta_1 - \delta_2|$.

D_{01}^{Φ}:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$

2a. $x_0 \leftarrow 0^n, b \leftarrow \{0,1\}$

2b. $rec_i \leftarrow rec_i^b$

2c. $\forall j \in \{1, \ldots, \ell\}$:

$m_j \leftarrow \Phi(x_{j-1})$
$r_j \leftarrow \mathsf{H}(rec_j)$
$x_j \leftarrow h(m_j, r_j)$

3. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$

4. **if** $\hat{b} = b$ **then** output 1 **else** output 0

D_{12}^{Φ}:

1. $i, rec_1, \ldots, rec_{i-1}, rec_{i+1}, \ldots, rec_\ell, rec_i^0, rec_i^1, a \leftarrow A_1$

2a. $x_0 \leftarrow 0^n, b \leftarrow \{0,1\}$

2b. $rec_i \leftarrow rec_i^b$

2c. $\forall j \in \{1, \ldots, \ell\}$:

if $x_{j-1} = x_{k-1}$ for some $k < j$ **then** $m_j \leftarrow m_k$
else $m_j \leftarrow \{0,1\}^n$
$r_j \leftarrow \mathsf{H}(rec_j)$
if $j = i$ **then** $x_j \leftarrow \Phi(r_i)$ **else** $x_j \leftarrow h(m_j, r_j)$

3. $\hat{b} \leftarrow A_2(a, x_1, \ldots, x_\ell, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_\ell)$

4. **if** $\hat{b} = b$ **then** output 1 **else** output 0

Fig. 4. The distinguishers.

The distinguisher D_{01}^{Φ} is construced (Fig. 4, left) so that in case of the oracle $h(IV, \cdot)$ it perfectly simulates Game_0 (the original attacking game), and in case of the random oracle $\Omega(\cdot)$ it perfectly simulates Game_1 (where random masks are used). Hence, $\mathsf{Adv}(D_{01}^{\Phi}) = |\delta_0 - \delta_1|$.

The other distinguisher D_{12}^{Φ} is constructed (Fig. 4, right) so that in case of the oracle $h(IV, \cdot)$ it perfectly simulates Game_1, whereas in case of the random oracle Ω it simulates Game_2 (the simulator $\mathcal{S}$).

Hence, we have $\mathsf{Adv}(D_{12}^{\Phi}) = |\delta_1 - \delta_2|$, and thereby,

$$\delta = 2|\delta_0 - \delta_2| \le 2\left(|\delta_0 - \delta_1| + |\delta_1 - \delta_2|\right) = 2\left(\mathsf{Adv}(D_{01}^{\Phi}) + \mathsf{Adv}(D_{12}^{\Phi})\right) \le 2\left(\frac{t_1}{S} + \frac{t_2}{S}\right) = 4\frac{t}{S},$$

and hence the log-signing scheme is $\frac{S}{4}$-secure IND-CPA content concealing. □

4 Reference Algorithms

In this section we present reference algorithms for aggregating a log block, extracting an integrity proof for an individual record, and verifying a record based on such proof. We also discuss some potential trade-offs where additional security benefits or runtime reductions could be gained at the cost of increased storage overhead.

4.1 Canonical Binary Trees

In the previous section we did not specify the shape of the Merkle tree. If the number of leaves is an even power of two, building a complete binary tree seems natural, but in other cases the appropriate shape is not necessarily obvious.

Of course, it is crucially important to build the tree in a deterministic manner so that the verifier would be able to construct the exact same tree as the signer did. Another consideration is that to achieve the logarithmic size of the integrity proofs of the individual records, the tree should not be overly unbalanced. Thus, we define the *canonical binary tree* with n leaf nodes (shown for $n = 11$ on Fig. 5) to be built as follows:

1. The leaf nodes are laid out *from left to right* (white nodes on the figure).
2. The leaf nodes are collected into complete binary trees *from left to right*, making each tree as big as possible using the leaves still available (adding the gray nodes on the figure).
3. The complete trees are merged into a single tree *from right to left* which means joining the two smallest trees on each step (adding the black nodes on the figure).

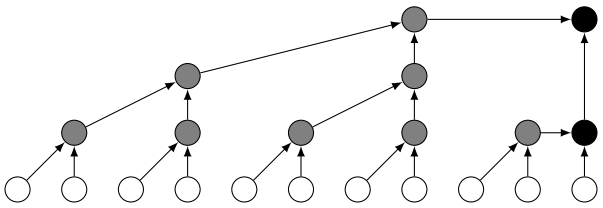

Fig. 5. Canonical binary tree with 11 leaves (white nodes), grouped into three complete trees (gray nodes), and merged to a single tree with minimal height (black nodes).

A useful property of canonical trees is that they can be built on-line, as the leaf nodes arrive, without knowing in advance the eventual size of the tree, and keeping in memory only logarithmic number of nodes (the root nodes of the complete binary trees constructed so far).

4.2 Aggregation of Log Records

Algorithm 1 aggregates a block of records into a canonical Merkle tree for signing or verification. The input description numbers the records $1 \ldots N$, but the value of N is not used and the algorithm can easily be implemented for processing the records on-line.

The algorithm also enforces the hash chain length limiting as mentioned in Sec. 2.1. This is done by assigning to each node a level value that strictly increases on any path from a leaf to a root and including that value among the data hashed at each step. To achieve this, the level of a leaf node is defined to be 1 and the level of a non-leaf node to be 1 more than the maximum of the levels of its child nodes.

An amortized constant number of hashing operations is needed per record and the worst-case actual processing time per record is logarithmic in the number of records in the block, as is the size of the auxiliary working memory needed.

To sign a log block, Alg. 1 could be used in the following manner:

1. A fresh random value is generated for IV.
2. The log records of the current block, the IV, and the last leaf hash value from the previous block are fed into Alg. 1.
3. The resulting root hash value is signed and the last leaf hash value from this block passed on to aggregation of the next block.
4. At the very least the IV and the signature on the root hash value must be saved for later verification.

Algorithm 1 Aggregate a block of records

inputs
$rec_{1\ldots N}$: input records
IV: initial value for the blinding masks
x_0: last leaf hash of previous block (zero for first block)
do
{Initialize block: create empty roots list}
$R :=$ empty list
{Process records: add to Merkle forest in order}
for $i := 1$ **to** N **do**
$r_i := \mathrm{hash}(rec_i)$
$m_i := \mathrm{hash}(x_{i-1}, IV)$
$x_i := \mathrm{hash}(m_i, r_i, 1)$
{Add x_i to the forest as new leaf, update roots list}
$t := x_i$
for $j := 1$ **to** $\mathrm{length}(R)$ **do**
if $R_j =$ **none then**
$R_j := t$; $t :=$ **none**
else if $t \neq$ **none then**
$t := \mathrm{hash}(R_j, t, j+1)$; $R_j :=$ **none**
if $t \neq$ **none then**
$R := R \mid\mid t$; $t :=$ **none**
{Finalize block: merge forest into a single tree}
$root :=$ **none**
for $j := 1$ **to** $\mathrm{length}(R)$ **do**
if $root =$ **none then**
$root := R_j$; $R_j :=$ **none**
else if $R_j \neq$ **none then**
$root := \mathrm{hash}(R_j, root, j+1)$; $R_j :=$ **none**
outputs
$root$: root hash of this block (to be signed or verified)
x_N: last leaf hash of this block (for linking next block)

Algorithm 2 Extract a hash chain for verifying one record

inputs
$rec_{1\ldots N}$: input records
pos: position of the object record within block ($1 \ldots N$)
IV: initial value for the blinding masks
x_0: last leaf hash of previous block (zero for first block)
do
{Initialize block}
$R :=$ empty list; $C :=$ empty list
$\ell :=$ **none** {object record not in any level yet}
{Process records, keeping track of the object}
for $i := 1$ **to** N **do**
$r_i := \mathrm{hash}(rec_i)$
$m_i := \mathrm{hash}(x_{i-1}, IV)$
$x_i := \mathrm{hash}(m_i, r_i, 1)$
if $i = pos$ **then**
$C := C \mid\mid (\textbf{right}, m_i, 0)$ {step to compute x_i}
$\ell := 1$; $d :=$ **right** {x_i be added as leaf to the right}
{Add x_i to the forest as new leaf}
$t := x_i$
for $j := 1$ **to** $\mathrm{length}(R)$ **do**
if $R_j =$ **none then**
if $j = \ell$ **then** $d :=$ **left**
$R_j := t$; $t :=$ **none**
else if $t \neq$ **none then**
if $j = \ell$ **then**
if $d =$ **right then** $S := R_j$ **else** $S := t$
$C := C \mid\mid (d, S, 0)$
$\ell := j+1$; $d :=$ **right**
$t := \mathrm{hash}(R_j, t, j+1)$; $R_j :=$ **none**
if $t \neq$ **none then**
if $\mathrm{length}(R) < \ell$ **then** $d :=$ **left**
$R := R \mid\mid t$; $t :=$ **none**
{Finalize block}
$root :=$ **none**
for $j := 1$ **to** $\mathrm{length}(R)$ **do**
if $root =$ **none then**
if $j = \ell$ **then** $d :=$ **right else**
$root := R_j$; $R_j :=$ **none**
else if $R_j \neq$ **none then**
if $j \geq \ell$ **then**
if $d =$ **right then** $S := R_j$ **else** $S := root$
$C := C \mid\mid (d, S, j - \ell)$
$\ell := j+1$; $d :=$ **right**
$root := \mathrm{hash}(R_j, root, j+1)$; $R_j :=$ **none**
outputs
C: hash chain from the object to the root of block

Algorithm 3 Re-compute the root hash value of a block

inputs
rec: input record
C: hash chain from the record to the root of block
do
$\ell := 0$
$root := \mathrm{hash}(rec)$
for $i := 1$ **to** $\mathrm{length}(C)$ **do**
$(d, S, L) := C_i$ {direction, sibling, level correction}
$\ell := \ell + L + 1$
if $d =$ **left then**
$root := \mathrm{hash}(root, S, \ell)$
else
$root := \mathrm{hash}(S, root, \ell)$
outputs
$root$: root hash of the block (for verification)

Given the above, the way to verify a signed log block is quite obvious:

1. The log records, the IV saved during signing, and the last leaf hash value from the previous block are fed into Alg. 1.
2. The freshly re-computed root hash value is verified against the saved signature.

A placeholder value filled with zeroes is used for the last leaf hash value of the previous block in the very first block of the log (when there is no previous block) or when there has been a discontinuity in the log (for example, when the logging service has been down).

Although not strictly required in theory, in practice the last leaf hash value of the previous log block should also be saved along with the IV and the signature. Otherwise the verification of the current block would need to re-hash the previous block to obtain the required input, which in turn would need to re-hash the next previous block, etc. While this would obviously be inefficient, an even more dangerous consequence would be that any damage to any log block would make it impossible to verify any following log blocks, as one of the required inputs for verification would no longer be available.

Considering the negative scenarios in more detail, the only conclusion that can be derived from a failed verification in the minimal case above would be that something has been changed in either the log block or the authentication data. If it is desirable to be able to detect the changes more precisely, either the record hash values r_i or the leaf hash values x_i computed by Alg. 1 could be saved along with the other authentication data. Then the sequence of hash values could be authenticated against the signature and each record checked against its hash value, at the expense of small per-record storage overhead.

It should also be noted that when the record hashes are saved, they should be kept with the same confidentiality as the log data itself, to prevent the informed brute-force attack discussed in Sec. 2.1.

4.3 Extraction of Hash Chains

Algorithm 2 extracts the hash chain needed to prove or verify the integrity of an individual record. The core is similar to Alg. 1, with additional tracking of the hash values that depend on the object record and collecting a hash chain based on that tracking.

The output value is a sequence of (*direction*, *sibling hash*, *level correction*) triples. The direction means the order of concatenation of the incoming hash value and the sibling hash value. The level correction value is needed to account for cases when two sub-trees of unequal height are merged and the node level value increases by more than 1 on the step from the root of the lower sub-tree to the root of the merged tree. (The step from the lower black node to the higher one on Fig. 5 is an example.)

Because Alg. 2 is closely based on Alg. 1, its performance is also similar and thus it falls somewhat short of our ideal of sub-linear runtime for hash chain extraction. We do not expect this to be a real issue in practice, as locating the records to be presented as evidence is typically already a linear-time task and thus reducing the proof extraction time would not bring a significant improvement in the total time.

Also note that the need to access the full log file in this algorithm is not a compromise of our confidentiality goals, as the extraction process is executed by the owner of the log file and only the relevant log records and the hash chains computed for them by Alg. 2 are shipped to outside parties.

However, it would be possible to trade space for time and extra confidentiality, if desired. At the cost of storing two extra hash values per record, logarithmic runtime could be achieved and the need to look at any actual records during the hash chain extraction could be removed.

Indeed, if the hash values from all the Merkle tree nodes (shown in gray on Fig. 1) were kept, the whole hash chain could be extracted without any new hash computations. If the values (all of the same fixed size) would be stored in the order in which they are computed as x_i and R_j in Alg. 1, each of them could be seeked to in constant time.

4.4 Computation of Hash Chains

Algorithm 3 computes the root hash value of the Merkle tree from which the input hash chain was extracted. The hash chain produced by Alg. 2 and the corresponding log record should be fed into Alg. 3, and the output hash value verified against the signature to prove the integrity of the record.

5 Implementation

In this section we outline some practical concerns regarding the implementation of the proposed scheme for signing `syslog` messages. Out of the many possible deployment scenarios we concentrate on signing the output directed to a text file on a log collector device. (See [6], Sec. 4.1 for more details.)

5.1 General Technical Considerations

Log File Rotation, Block Size. In Sec. 2 we modeled the log as an ordered sequence of blocks, where each block in turn is an ordered sequence of a finite number of records, and noted that the case of `syslog` output being sent to a periodically rotated log file could be viewed as an instantiation of this model.

We now refine the model to distinguish the logical blocks (implied by signing) from the physical blocks (the rotated files), because it is often desirable to sign the records in a finer granularity than the frequency of file rotation.

A log file could contain several signed blocks, but for file management reasons, a signed block should be confined to a single file. This means that when log files are rotated, the current signature block should always be closed and a new one started from the beginning of the new file.

The hash links from the last record of previous block to the first record of the next block should span the file boundaries, though, to enable verification of the integrity of the whole log, however the files may have been rotated.

Implementations could support limiting the block sizes both by number of records and time duration. When a limit on the number of records is set and a block reaches that many records, it should be signed and a new block started. Likewise, when a duration limit is set and the oldest record in a block reaches the given age, the block should be signed and a new one started.

When both limits are set, reaching either one should cause the block to be closed and also reset both counters. Applying both limits could then be useful for systems with uneven activity. In this case the size limit would prevent the blocks growing too big during busy times and the time limit would prevent the earlier records in a block staying unsigned for too long during quiet times.

When neither limit is given, each block would cover a whole log file, as rotating the output files should close the current block and start a new one in any case.

Record Canonicalization. When log records are individually hashed for signing before they are saved to the output file, it is critical that the file could be unambiguously split back into records for verification. End-of-line markers are used as record separators in text-based `syslog` files and then multi-line records could not be recovered correctly unless the line breaks within the records would be escaped.

Often `syslog` implementations also escape various other control characters and in particular the ASCII NUL characters in the log records. Assuming that such escaping has been done either in the log source before the records are transmitted to the collector or in the collector before signing, the following rules for handling signed `syslog` files would ensure correct behavior:

- For signing, each record is hashed as-is.
- When records are written to a plain text file, each end-of-line within a record is escaped in some unambiguously reversible manner.
- When reading from a plain text file, each end-of-line is treated as a record separator (not part of either of the records it separates) and within the records the escaping done at writing time is reversed.
- For verification, the original records (with the record-separating end-of-line markers removed and escape sequences decoded) are hashed again.

Signature Technologies. Once the log blocks are aggregated, the root hash values have to be protected from future modifications. While it could seem natural to sign them using a standard public-key signature such as an OpenPGP [5] or PKCS#7 [9, 8] one, these offline signing technologies do not provide good forward security in case the logging server is compromised or privileged insiders abuse their access. An attacker could modify the logs and then re-hash

and re-sign all the blocks starting from the earliest modified one, as the signing keys would be available on the log collector host.

A cryptographic time-stamping service [2] could be used as a mitigation. Note that a time-stamp generally only provides evidence of the time and integrity of the time-stamped datum, but not the identity of the requesting party, so a time-stamp alone would not be sufficient to prevent a log file from another system being submitted instead of the original one. Therefore, time-stamps should be used in addition to, not in place of, signatures.

An alternative would be to use the OpenKSI keyless signatures [3] that combine the hash value of the signed datum, the identity of the requesting system, and the signing time into one independently verifiable cryptographic token. As verification of keyless signatures does not rely on secrecy of keys and does not need trusted third parties, they are particularly well-suited for signing logs with long-term evidentiary value.

5.2 Case Study: `rsyslog` Integration

The proposed log signing scheme has been implemented in `rsyslog`, a popular logging server implementing the `syslog` protocol and included as a standard component in several major Linux distributions.

Architecture. The `rsyslog` server has a modular architecture, as shown on Fig. 6, with a number of input and output modules available to receive log messages from various sources and store logs using different formats and storage engines.

Log signing has been implemented as a new optional functionality in the `omfile` output module that stores log data in plain text files. When signing is enabled, the signatures and related helper data items are stored in a binary file next to the log file. As the data is a sequence of non-human-readable binary tokens (hash values and signatures), there would be no benefit in keeping it in a text format.

Both files are needed for verification of log integrity. It is the responsibility of the log maintenance processes to ensure that the files do not get separated when the log files are archived or otherwise managed.

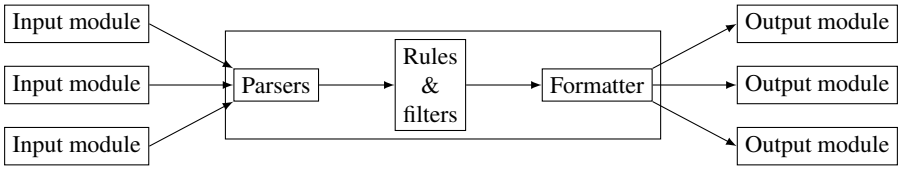

Fig. 6. General architecture of `rsyslog`.

Configuration. The `rsyslog` configuration is specified as a number of sets of rules applied to incoming messages. The minimal example shown on Fig. 7 configures the server to listen for incoming messages on TCP port 514 and apply the `perftest` rules to all received messages. The rules in turn just write all the records to the specified text file with no filtering or transformations.

The following signing-related configuration options have been added to the `omfile` output module:

- `sig.hashFunction`: the hash function to be used for record hashing and aggregation; currently the SHA-2 family, as well as the SHA-1 and RIPEMD-160 functions are supported; default is 256-bit SHA-2;
- `sig.block.sizeLimit`: upper limit on the number of records in a signed block, as discussed in Sec. 5.1; default is unlimited;
- `sig.keepRecordHashes`: whether to save the hash values of individual records; the benefit of keeping the values was discussed in Sec. 4.2; off by default;
- `sig.keepTreeHashes`: whether to save the intermediate hash values from the Merkle tree; the potential benefit of keeping the values was discussed in Sec. 4.3; off by default;

– `sig.provider`: name of the signature provider; this causes the specified provider module to be loaded and the signing functionality to be enabled for the corresponding output file; currently only the `gt` provider is implemented using the Guardtime time-stamping service that produces time-stamps signed with keyless signatures and could be seen as a hybrid between the X.509 time-stamping and OpenKSI signing options discussed in Sec. 5.1;
– `sig.timestampService`: the URL of the service to use; a public test service is used as default; note that the public service is anonymous and returns tokens that do not contain the identity of the requesting system.

```
module(load="imtcp")
input(type="imtcp" port="514" ruleset="perftest")

ruleset(name="perftest"){
    action(type="omfile" file="/var/log/signedtcp.log" sig.provider="gt")
}
```

Fig. 7. Minimal `rsyslog` configuration file with the log signing functionality enabled.

Performance. We tested the performance of the implementation to find out the overhead and possible bottlenecks. Testing was performed on 2011-era non-virtualized server hardware, with a quad-core Intel Xeon E5606 CPU. We used 64-bit CentOS 6.4 operating system and installed `rsyslog` version 7.3.15-2 from the Adiscon package repository. There was excess of memory and I/O resources, in all tests the performance was CPU-bound. Load was generated using the `loggen` utility which is part of the `syslog-ng 3.5.0 alpha0` package, another `syslog` server implementation.

We used TCP socket for logging in order to avoid potential message loss and mimic a real-life scenario with central logging server. We note that UDP and kernel device interface gave comparable results. The `rsyslog` configuration was as shown on Fig. 7: simplest possible, without any message processing.

Without signing we achieved sustained logging rate of approximately 400,000 messages per second. At this point the `rsyslog` input thread saturated one CPU core. Multiple input threads and multiple main queue worker threads allowed us to achieve slightly better performance. Here and below the average log message size was 256 bytes, and the default SHA2-256 hash algorithm was used in tests where signing was enabled.

Signed message logging rate was constantly higher than 100,000 messages per second. The limiting factor was the main queue worker thread which saturated one CPU core. For one signed output file the current signing scheme does not allow the building of the hash tree to be parallelized in an efficient way, because the order of the log messages must be preserved. Although possible to configure, multiple parallel main worker queues would spend most of their time waiting for synchronization and total signing performance would be inferior.

Storage of the record hash values (via `sig.keepRecordHashes`) and the intermediate Merkle tree hash values (via `sig.keepTreeHashes`) did not affect the signing performance significantly. Also, choosing different hash algorithms (via `sig.hashFunction`) did not have a significant impact.

Aggregating a log message incurs approximately three hash algorithm invocations. Considering that one CPU core can perform roughly one million SHA2-256 calculations with 64-byte input (as measured with `openssl speed sha256` command benchmarking the cryptographic library used to implement the `gt` module), the log signing performance achieved is reasonably close to optimal.

It should also be noted that the four-fold decrease of throughput from 400,000 messages per second to 100,000 messages per second is extreme, as in the baseline scenario no CPU power was spent on filtering the records.

Storage overhead depends on whether the record and tree hashes are stored, hash algorithm output size, and signature block size. In case of 256-byte log records and 32-byte hash values, the storage overhead is about 12% for keeping the record hashes and about 25% for keeping the tree hashes. The storage overhead caused by signatures themselves depends on the size of a log bloks, but is negligible in practical scenarios.

6 Conclusions

We have proposed a log signing scheme with good security properties and low overhead in both computational and storage resources. The integrity of either the whole log or any record can be verified, and in the latter case also a compact proof produced without leaking any information about the contents of other records. The scheme also allows for some flexibility in providing additional verification features and proof extraction performance gains at the cost of increased storage overhead, as listed in Table 1.

Table 1. Storage, runtime and verification feature trade-offs (N is log block size).

Characteristic	No hashes kept	Record hashes kept	Tree hashes kept
		Signing	
Per-record storage	none	1 hash value	2 hash values
Per-record computation	3 hashings	3 hashings	3 hashings
Per-block storage	1 signature value	1 signature value	1 signature value
Per-block computation	1 signing	1 signing	1 signing
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
		Whole log verification	
Report granularity	block	record	record
Time	$O(N)$	$O(N)$	$O(N)$
Memory	$O(\log N)$	$O(\log N)$	$O(\log N)$
		Record proof extraction	
Per-record storage	$O(\log N)$*	$O(\log N)$*	$O(\log N)$*
Time	$O(N)$	$O(N)$	$O(\log N)$
Memory	$O(\log N)$	$O(\log N)$	$O(1)$
		Record proof verification	
Report granularity	record	record	record
Time	$O(\log N)$	$O(\log N)$	$O(\log N)$
Memory	$O(1)$	$O(1)$	$O(1)$

* Asymptotically it's $O(\log N)$, but in many practical cases the $O(1)$ signature size dominates over the $O(\log N)$ hash chain size. For example, for the approximately 3600-byte signatures and 32-byte hash values used in our case study, the signature size exceeds the hash chain size for all $N < 2^{100}$.

An interesting direction for future development would be to extend the scheme for use in configurations where the log is signed at a source device, transported over one or more relay devices and then stored at a collector device. The current scheme would be usable only under the condition that the relay devices do not perform any filtering of the records, which is rarely the case in practice.

References

1. R. Accorsi. BBox: a distributed secure log architecture. In *Proceedings of the 7th European Conference on Public Key Infrastructures, Services and Applications*, pages 109–124. Springer, 2011.
2. C. Adams, P. Cain, D. Pinkas, and R. Zuccherato. Internet X.509 public key infrastructure time-stamp protocol (TSP). IETF RFC 3161, 2001.
3. A. Buldas, A. Kroonmaa, and A. Park. OpenKSI digital signature format. OpenKSI, 2012.
4. A. Buldas and M. Saarepera. On provably secure time-stamping schemes. In *Advances in Cryptology—ASIACRYPT 2004*, pages 500–514. Springer, 2004.
5. J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. IETF RFC 4880, 2007.
6. R. Gerhards. The syslog protocol. IETF RFC 5424, 2009.
7. J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research*, pages 203–211. Australian Computer Society, 2006.

8. R. Housley. Cryptographic message syntax (CMS). IETF RFC 5652, 2009.
9. B. Kaliski. PKCS#7: Cryptographic message syntax version 1.5. IETF RFC 2315, 1998.
10. J. Kelsey, J. Callas, and A. Clemm. Signed syslog messages. IETF RFC 5848, 2010.
11. D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage*, 5(1):2:1–2:21, 2009.
12. R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134. IEEE Computer Society, 1980.
13. B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems Security*, 2(2):159–176, 1999.
14. V. Stathopoulos, P. Kotzanikolaou, and E. Magkos. A framework for secure and verifiable logging in public communication networks. In *Proceedings of the First International Conference on Critical Information Infrastructures Security*, pages 273–284. Springer, 2006.