Time-Scoped Searching of Encrypted Audit Logs
(Extended Abstract)

Darren Davis[1], Fabian Monrose[1], and Michael K. Reiter[2]

[1] Johns Hopkins University, Baltimore, MD, USA
ddavis@jhu.edu, fabian@cs.jhu.edu
[2] Carnegie Mellon University, Pittsburgh, PA, USA
reiter@ece.cmu.edu

Abstract. In this paper we explore restricted delegation of searches on encrypted audit logs. We show how to limit the exposure of private information stored in the log during such a search and provide a technique to delegate searches on the log to an investigator. These delegated searches are limited to authorized keywords that pertain to specific time periods, and provide guarantees of completeness to the investigator. Moreover, we show that investigators can efficiently find all relevant records, and can authenticate retrieved records without interacting with the owner of the log. In addition, we provide an empirical evaluation of our techniques using encrypted logs consisting of approximately 27,000 records of IDS alerts collected over a span of a few months.

1 Introduction

In this paper we present a protocol by which Alice can delegate to an investigator the ability to search an audit log on her server for specific keywords generated in specific time periods. Following this delegation, the investigator can perform that search so that the server, even if corrupted by an attacker after the time periods being searched (but before the search itself), cannot undetectably mislead the investigator. The investigator, however, is limited to precisely the time periods and keywords for which Alice delegated searching authority, and gains no information for other time periods and other keywords.

In our model, Alice is trusted to always protect her secrets and follow the protocols of the system. To help justify this trust assumption, Alice remains somewhat isolated and communication with her is infrequent. Specifically, Alice interacts with the server to establish the log and to periodically update keys at the server, and with the investigator to delegate rights to search the log; otherwise Alice is not involved in the logging or investigative protocols. Unlike Alice, the server is not fully trusted and may be compromised at any point in time. Until then, it diligently maintains an audit log of events that have occurred on the server. Similarly, the investigator is not trusted: Alice and the server take measures to ensure that the investigator gleans no further information about the log beyond that permitted by Alice.

J. López, S. Qing, and E. Okamoto (Eds.): ICICS 2004, LNCS 3269, pp. 532–545, 2004.

While there are existing proposals that separately implement audit logs on untrusted servers, searches on encrypted data by keywords, and time-based cryptographic primitives, we believe our scheme is the first that integrates all of these constructions in an efficient manner. Moreover, we propose several new ideas which allow for a practical implementation of our goals. We detail a real implementation of our scheme and evaluate its performance using logs from a Snort intrusion detection system [6] comprised of roughly 27,000 alerts and includes attacks from nearly 1700 distinct IP address during a span of a few months.

The rest of this paper is organized as follows. In Section 2 we introduce some preliminaries and formally outline our requirements. Section 3 examines related work. Section 4 introduces our logging, authorization and searching protocols. We provide a security evaluation in Section 5. Implementation details and empirical results are presented in Section 6.

2 Preliminaries

For the purposes of this paper, a log is simply a sequence of *records*, each of which encodes a *message* m. Time is broken into non-overlapping intervals, or *time periods*, each with an index typically denoted by p (i.e., T_p is an interval of time, followed by interval T_{p+1}). If R is a record then $\mathsf{time}(R)$ denotes the index of the time period when R was written, and $\mathsf{words}(R) = \{w_1, \ldots, w_n\}$ denotes a set of keywords that characterize its message m, and on which searching will be possible. Different records can have different numbers of keywords.

Our solution to the secure logging problem consists of three protocols:

- $\mathsf{LogIt}(m, W)$: This protocol runs locally on the server. It takes as input a message m and a collection W of words. ($|W| = n$ need not be fixed, but can vary per invocation.) It creates a log record R that encodes m.
- $\mathsf{Auth}(T, W)$: This protocol runs between Alice and the investigator. The investigator initiates this protocol with past time periods $T = [p_1, p_k]$ (i.e., a contiguous sequence of k periods indexed $p_1, \ldots, p_k$) and words W, and interacts with Alice to obtain authority (i.e., trapdoors) to search for log records R such that $\mathsf{time}(R) \in T$ and $W \cap \mathsf{words}(R) \neq \emptyset$. Upon completion, this protocol indicates either $\mathsf{granted}$ or denied to the investigator.
- $\mathsf{Access}(T, W)$: This protocol runs between the investigator and the server. The investigator begins this protocol with past time periods $T = [p_1, p_k]$ and words W. Using this protocol, the investigator retrieves from the server each message m of each record R such that $\mathsf{time}(R) \in T$ and $W \cap \mathsf{words}(R) \neq \emptyset$. This protocol returns to the investigator either a set of messages $\{m\}$ or an error code $\mathsf{tampered}$.

First consider an attacker who compromises the server only, and let p_{comp} denote the time period in which the server is compromised. If $T = [p_1, p_k]$, we abuse notation by using $T < p$ as shorthand for $p_k < p$. We require a number of properties of a secure logging scheme, which we detail below. Let $\mathsf{match}(T, W) = \{m : \mathsf{LogIt}(m, W')$ occurred in some $p \in T \wedge W' \cap W \neq \emptyset)\}$.

Requirement 1 (Liveness) *If* $\mathsf{Access}(T, W)$ *is executed in time period* p *and returns* tampered*, then* $p \geq p_{\mathsf{comp}}$.

Requirement 2 (Soundness) *If* $T < p_{\mathsf{comp}}$*, then either* $\mathsf{Access}(T, W) =$ tampered *or* $\mathsf{Access}(T, W) \subseteq \mathsf{match}(T, W)$.

Requirement 3 (Completeness) *If* $T < p_{\mathsf{comp}}$*, then either* $\mathsf{Access}(T, W) =$ tampered *or* $\mathsf{match}(T, W) \subseteq \mathsf{Access}(T, W)$.

Informally, Liveness simply requires that Access not return tampered until after a compromise occurs. Soundness and Completeness require that an adversary who compromises the server be unable to undetectably insert or delete log records, respectively. We now add an additional property to protect against an investigator who attempts to overstep his authority granted by Alice.

Requirement 4 (Privacy) *Let* $\mathsf{Auth}(T_1, W_1), \ldots, \mathsf{Auth}(T_k, W_k)$ *denote all* Auth *protocols that have resulted in* granted *and such that* $T_i < p_{\mathsf{comp}}$*. Then, the investigator can learn only what it can infer from* $\mathsf{match}(\cup_i T_i, \cup_i W_i)$.

In addition to these properties, we include efficiency in our goals and measure quality of each of the LogIt, Auth and Access protocols according to their computation and message complexities.

3 Related Work

Several works [16, 9, 3, 7] consider techniques for searching encrypted documents stored on a server for any word in the document. Typically, symmetric cryptosystems are used to encrypt documents that are then transferred to a server where searches are subsequently performed. These work differs from our setting in that the server itself performs the search, and returns relevant results to the document's owner. In the context of an audit log, since the server creates and encrypts log entries that must be eventually searched and decrypted by investigators, symmetric cryptosystems are not suitable for this purpose – once the server is compromised an adversary would obtain all of the keying material. Boneh *et al.* [4] provide a formal definition for a more relevant searchable public key-encryption scheme, but their constructions only allow for matching documents to be identified by the searcher.

In [15] Schneier *et al.* propose a scheme for securing audit logs on a remote server. The authentication of log entries is accomplished using a hash chain and a linearly evolving MAC. To facilitate searching, records contain type information, and the the owner of the log allows a searcher to read selected records based on type information of those records. However, the approach is inefficient as it requires one key to be sent from the owner to the searcher for *each* record to be searched, and the searcher must retrieve *all* records from the server to verify the hash chain. The scheme is improved upon in [12] by using a tree based scheme for deriving keys in which a non-leaf key can be used to derive all of the keys in its subtree. Unfortunately, to remain secure and efficient, only a limited set of

record types are allowed. This limitation is problematic for an audit log, where the extracted keywords may include, for example, an IP address.

Waters *et al.* [17] subsequently proposed a searchable audit log scheme similar to the tamper resistant audit log schemes of [15]. There, in order to add a message m_i with a set of keywords to the log, a random key K_i is chosen to encrypt m_i. K_i is identity-based encrypted [5] using each keyword as a public key. Similar to [15] each record also contains part of a hash chain and a corresponding MAC. To perform a search for keyword w, an investigator requests the trapdoor for that keyword (d_w) from the owner. Next, she retrieves each record and attempts to decrypt each word using d_w. To avoid retrieving all records and attempting one IBE operation per keyword [17] suggests an improvement that groups a small number of consecutive log records into a "block". Once all of the records in the block are stored, a special index record is written; for each keyword w associated with any record in the block, the index record contains an IBE with public key w of a list of records that contain w and the corresponding encryption keys. Unfortunately, as we show in Section 6, this enhancement still has significant performance limitations.

More distantly related work is that of time-based signatures (e.g, [11, 13]). For the most part, these work present schemes for timestamping digital documents using trusted servers. In some cases, e.g [13], identity-based constructions are used to reveal signatures on documents (or the documents themselves) at a specified point of time in the future; documents are encrypted with a public key for the date on which they are to be revealed, and each day, the server reveals the IBE decryption key corresponding to that date.

Discussion. The approach of [17] does not meet the *soundness* requirement as searches cannot be time-restricted – hence Access would return all entries that match irrespective of when these records were written. For similar reasons, their approaches for searching encrypted audit logs do not meet the *Privacy* requirement. Note that using the tree-based approach of [12] to encrypt log or index records, one can adapt [17] to achieve time limited searches. However, such an approach would be inefficient. To see why, consider how time-scoped searching could be trivially achieved in [17]. To do so, the time $\mathsf{time}(R_i)$ during which a record R_i occurs is stored in that record and identity-based encrypted using $\mathsf{time}(R_i)|w_{ij}$ instead of w_{ij}. Searches are performed as before, with the only change being that the investigator asks Alice for the trapdoors corresponding to each keyword she wants to search for during an interval of time. Unfortunately, this simplistic scheme results in a significant performance penalty for Alice; to search for a single keyword in t distinct time zones, Alice must provide the investigator with $O(t)$ IBE keys.

4 Protocols

In this section we detail the logging algorithm executed on Alice's computer, as well as the authorization and access protocols that are executed by the investigator. Our algorithms utilize a number of cryptographic tools:

Symmetric Encryption. A symmetric encryption system is a tuple $(\mathcal{K}^{\mathsf{sym}}, \mathcal{M}^{\mathsf{sym}}, \mathcal{C}^{\mathsf{sym}}, \mathcal{E}^{\mathsf{sym}}, \mathcal{D}^{\mathsf{sym}})$ where $\mathcal{K}^{\mathsf{sym}}$, $\mathcal{M}^{\mathsf{sym}}$ and $\mathcal{C}^{\mathsf{sym}}$ are sets of keys, plaintexts, and ciphertexts, respectively. $\mathcal{E}^{\mathsf{sym}}$ is a randomized algorithm that on input $K \in \mathcal{K}^{\mathsf{sym}}$ and $M \in \mathcal{M}^{\mathsf{sym}}$ outputs a $C \in \mathcal{C}^{\mathsf{sym}}$; we write this $C \leftarrow \mathcal{E}^{\mathsf{sym}}_K(M)$. $\mathcal{D}^{\mathsf{sym}}$ is a deterministic algorithm that on input $K \in \mathcal{K}^{\mathsf{sym}}$ and $C \in \mathcal{C}^{\mathsf{sym}}$ outputs $M \in \mathcal{M}^{\mathsf{sym}}$, i.e., $M \leftarrow \mathcal{D}^{\mathsf{sym}}_K(C)$, or else outputs $\perp$ (if C is not a valid ciphertext for key K). Naturally we require that if $C \leftarrow \mathcal{E}^{\mathsf{sym}}_K(M)$ then $M \leftarrow \mathcal{D}^{\mathsf{sym}}_K(C)$. In addition, we require that $(\mathcal{K}^{\mathsf{sym}}, \mathcal{M}^{\mathsf{sym}}, \mathcal{C}^{\mathsf{sym}}, \mathcal{E}^{\mathsf{sym}}, \mathcal{D}^{\mathsf{sym}})$ be secure under chosen ciphertext attacks, i.e., property ROR-CCA from [1].

Identity-based Encryption. An identity-based encryption system is a tuple $(\mathcal{M}^{\mathsf{ibe}}, \mathcal{C}^{\mathsf{ibe}}, \mathcal{I}^{\mathsf{ibe}}, \mathcal{X}^{\mathsf{ibe}}, \mathcal{E}^{\mathsf{ibe}}, \mathcal{D}^{\mathsf{ibe}})$ where $\mathcal{M}^{\mathsf{ibe}}$ and $\mathcal{C}^{\mathsf{ibe}}$ are sets of plaintexts and ciphertexts, respectively. $\mathcal{I}^{\mathsf{ibe}}$ is an initialization routine that generates a "master key" K^{ibe} (and other public parameters). $\mathcal{X}^{\mathsf{ibe}}$ is an algorithm that, on input $w \in \{0,1\}^*$ and K^{ibe}, outputs a "private key" d, i.e., $d \leftarrow \mathcal{X}^{\mathsf{ibe}}_{K^{\mathsf{ibe}}}(w)$. $\mathcal{E}^{\mathsf{ibe}}$ is a randomized algorithm that on input $w \in \{0,1\}^*$ and $M \in \mathcal{M}^{\mathsf{ibe}}$ outputs a $C \in \mathcal{C}^{\mathsf{ibe}}$; we write this $C \leftarrow \mathcal{E}^{\mathsf{ibe}}_K(M)$. $\mathcal{D}^{\mathsf{ibe}}$ is an algorithm that with input a private key d and ciphertext $C \in \mathcal{C}^{\mathsf{ibe}}$, outputs an $M \in \mathcal{M}^{\mathsf{ibe}}$, i.e., $M \leftarrow \mathcal{D}^{\mathsf{ibe}}_d(C)$, or else outputs $\perp$. We require that if $d \leftarrow \mathcal{X}^{\mathsf{ibe}}_{K^{\mathsf{ibe}}}(w)$ and $C \leftarrow \mathcal{E}^{\mathsf{ibe}}_w(M)$ then $M \leftarrow \mathcal{D}^{\mathsf{ibe}}_d(C)$. We require that $(\mathcal{M}^{\mathsf{ibe}}, \mathcal{C}^{\mathsf{ibe}}, \mathcal{I}^{\mathsf{ibe}}, \mathcal{X}^{\mathsf{ibe}}, \mathcal{E}^{\mathsf{ibe}}, \mathcal{D}^{\mathsf{ibe}})$ is secure under chosen ciphertext attacks, i.e., property IND-ID-CCA from [5].

Hash functions. We use a deterministic hash function $h : \{0,1\}^* \rightarrow \mathsf{Range}(h)$, which we model as a random oracle [2]. Since our protocol utilizes outputs of h as cryptographic keys for symmetric encryption, we require $\mathsf{Range}(h) = \mathcal{K}^{\mathsf{sym}}$.

Digital signatures. A digital signature algorithm is a tuple $(\mathcal{M}^{\mathsf{sig}}, \mathsf{Pub}^{\mathsf{sig}}, \mathsf{Priv}^{\mathsf{sig}}, \Sigma^{\mathsf{sig}}, \mathcal{I}^{\mathsf{sig}}, \mathcal{S}^{\mathsf{sig}}, \mathcal{V}^{\mathsf{sig}})$ where $\mathcal{M}^{\mathsf{sig}}$, $\mathsf{Pub}^{\mathsf{sig}}$, $\mathsf{Priv}^{\mathsf{sig}}$ and Σ^{sig} are sets of messages, public verification keys, private signature keys, and signatures, respectively. $\mathcal{I}^{\mathsf{sig}}$ is a randomized algorithm that produces a signature key $S \in \mathsf{Priv}^{\mathsf{sig}}$ and corresponding verification key $V \in \mathsf{Pub}^{\mathsf{sig}}$, i.e,. $(S, V) \leftarrow \mathcal{I}^{\mathsf{sig}}$. $\mathcal{S}^{\mathsf{sig}}$ is an algorithm that on input $S \in \mathsf{Priv}^{\mathsf{sig}}$ and message $M \in \mathcal{M}^{\mathsf{sig}}$, returns a signature $\sigma \in \Sigma^{\mathsf{sig}}$, i.e., $\sigma \leftarrow \mathcal{S}^{\mathsf{sig}}_S(M)$. $\mathcal{V}^{\mathsf{sig}}$ is an algorithm that on input $V \in \mathsf{Pub}^{\mathsf{sig}}$, message $M \in \mathcal{M}^{\mathsf{sig}}$, and signature $\sigma \in \Sigma^{\mathsf{sig}}$, returns a bit. We require that if $(S, V) \leftarrow \mathcal{I}^{\mathsf{sig}}$ and $\sigma \leftarrow \mathcal{S}^{\mathsf{sig}}_S(M)$, then $\mathcal{V}^{\mathsf{sig}}_V(M, \sigma) = 1$. We require that the signature scheme is existentially unforgeable against chosen message attacks [10].

Tuples. We treat protocol messages as typed tuples, denoted $\langle \ldots \rangle$ for tuples of multiple elements; we will typically drop the brackets for a tuple of one element. All tuples are "typed" in the sense that the set from which the element is drawn is explicitly named in the tuple representation, i.e., a tuple $\langle i, C \rangle$ where $i \in \mathbb{N}$ and $C \in \mathcal{C}^{\mathsf{ibe}}$ would be represented as $(i : \mathbb{N}, C : \mathcal{C}^{\mathsf{ibe}})$. We presume that all algorithms confirm the types of their arguments and each tuple they process, particularly those formed as the result of decryption. For readability, however, we do explicitly include these checks in our protocol descriptions.

Below we detail the LogIt, Auth and Access protocols. Alice initializes these protocols by generating a master key for the identity based encryption scheme, i.e., $K^{\mathsf{ibe}} \leftarrow \mathcal{I}^{\mathsf{ibe}}()$, which she keeps secret. She provides to the server the public parameters of the identity-based encryption scheme and any other public parameters needed by the cryptographic algorithms.

4.1 Logging

The logging function LogIt that is executed on the server is shown in Figure 1. This figure contains three groups of lines: global variables (lines 1–7), the LogIt function itself (lines 8–13), and another function AnchorIt (lines 14–24) that will be explained subsequently.

The LogIt function is given the message m to be logged and words $w_1, \ldots, w_n$ that characterize this message (line 8), and is required to insert this message as the next record in the log. The log itself is the global variable record (line 1) that maps natural numbers to log records (which themselves are elements of $\mathsf{Range}(h) \times \mathcal{C}^{\mathsf{sym}}$), and the index of the position in which this new record should be placed is rIndex (line 2).

The LogIt function begins by assembling the plaintext record R consisting of rIndex, m, and one "backpointer" $\mathsf{bp}(w_j)$ for each $j \in [1, n]$ (line 9). Each backpointer is an element of $\mathcal{C}^{\mathsf{ibe}}$ generated by encrypting with the word w_j. The plaintext of the backpointer $\mathsf{bp}(w_j)$ indicates the record created by the most recent previous LogIt invocation in which w_j was among the words provided as an argument.

The actual plaintext of $\mathsf{bp}(w_j)$ becomes apparent when considering how the current invocation of LogIt updates each $\mathsf{bp}(w_j)$ to point, in effect, to the record this invocation is presently creating: it simply encrypts $h(R)$ under w_j (line 12). As a mechanism to speed up other protocols that will be described later, the plaintext also includes the index j. Once $h(R)$ and j are encrypted under w_j, they are saved for use in the next invocation of LogIt (or AnchorIt, see below).

R itself is stored in record(rIndex) after encrypting it with $h(R)$ and prepending $h(h(R))$ (line 13). This counter-intuitive construction is justified by the following observation: $r = h(R)$ can serve as a search key for requesting this record (i.e., by asking the server for the record with first component $h(r)$); the decryption key for decrypting its second component $\mathcal{E}^{\mathsf{sym}}_r(R)$; and a means to authenticate the result R by checking that $h(R) = r$. The utility of this construction will become clearer below. We note that this construction requires h to behave like a random oracle, as it must return a random encryption key from $\mathcal{K}^{\mathsf{sym}}$.

Periodically, the server invokes the AnchorIt routine (line 15) to create an *anchor record*. Anchor records, each of which is an element of $\mathsf{Range}(h) \times \mathcal{C}^{\mathsf{sym}} \times \Sigma^{\mathsf{sig}}$, are stored in anchor (line 3), and the next anchor record will be written to anchor(aIndex). Intuitively, the duration of time that transpires between writing anchor records defines the minimum duration of time in which an investigator can be granted authority to search.

The AnchorIt function is called with a *time key* $TK \in \mathcal{K}^{\mathsf{sym}}$, and an *authentication key* $AK \in \mathsf{Priv}^{\mathsf{sig}}$. The AnchorIt function creates a record A (line 17) that

1. $\mathsf{record} : \mathbb{N} \rightarrow \mathsf{Range}(h) \times \mathcal{C}^{\mathsf{sym}}$ - the log records
2. $\mathsf{rIndex} : \mathbb{N}$ - last record written
3. $\mathsf{anchor} : \mathbb{N} \rightarrow \mathsf{Range}(h) \times \mathcal{C}^{\mathsf{sym}} \times \Sigma^{\mathsf{sig}}$ - anchor records; written periodically
4. $\mathsf{aIndex} : \mathbb{N}$ - last anchor written
5. $\mathsf{prevAncHash} : \mathsf{Range}(h)$ - hash of plaintext of previous anchor; initialized by $\mathsf{prevAncHash} \stackrel{R}{\leftarrow} \mathsf{Range}(h)$
6. $\mathsf{current} : \{0,1\}^* \rightarrow \{\mathtt{true}, \mathtt{false}\}$ - `true` iff w used since last anchor; initialized $\mathsf{current}(w) = \mathtt{false}$
7. $\mathsf{bp} : \{0,1\}^* \rightarrow \mathcal{C}^{\mathsf{ibe}}$ - plaintext of $\mathsf{bp}(w)$ contains hash of last record pertaining to w; initialized with $r \stackrel{R}{\leftarrow} \mathcal{M}^{\mathsf{ibe}}; \mathsf{bp}(w) \leftarrow \mathcal{E}^{\mathsf{ibe}}_w(r)$

LogIt function, invoked locally to generate new log record

8. $\mathsf{LogIt}(m, w_1, \ldots, w_n : \{0,1\}^*)$ - m is message to be logged; $w_1, \ldots, w_n$ is
9. $\quad R \leftarrow \langle \mathsf{rIndex}, m, \mathsf{bp}(w_1), \ldots, \mathsf{bp}(w_n) \rangle$ vector of words describing m
10. $\quad$ **foreach** $j \in [1, n]$
11. $\quad\quad \mathsf{current}(w_j) \leftarrow \mathtt{true}$ - mark w_j involved in a record
12. $\quad\quad \mathsf{bp}(w_j) \leftarrow \mathcal{E}^{\mathsf{ibe}}_{w_j}(\langle h(R), j \rangle)$ - *next* record with w_j will include $\mathsf{bp}(w_j)$; anyone able to decrypt $\mathsf{bp}(w_j)$ can authenticate R using $h(R)$ (c.f., line 13)
13. $\quad \mathsf{record}(\mathsf{rIndex}) \leftarrow \langle h(h(R)), \mathcal{E}^{\mathsf{sym}}_{h(R)}(R) \rangle$ - $h(R)$ permits record to be retrieved,
14. $\quad \mathsf{rIndex} \leftarrow \mathsf{rIndex} + 1$ decrypted and authenticated

AnchorIt function, invoked locally to write an anchor record

15. $\mathsf{AnchorIt}(TK : \mathcal{K}^{\mathsf{sym}}, AK : \mathsf{Priv}^{\mathsf{sig}})$
16. $\quad w_1, \ldots, w_n \leftarrow \langle w : \mathsf{current}(w) = \mathtt{true} \rangle$ - words in log records since last anchor
17. $\quad A \leftarrow \langle \mathsf{aIndex}, \mathsf{prevAncHash}, \mathsf{bp}(w_1), \ldots, \mathsf{bp}(w_n) \rangle$
18. $\quad \mathsf{prevAncHash} \leftarrow h(A)$
19. $\quad$ **foreach** $j \in [1, n]$
20. $\quad\quad \mathsf{current}(w_j) \leftarrow \mathtt{false}$ - reset $\mathsf{current}(w)$
21. $\quad\quad C \leftarrow \mathcal{E}^{\mathsf{sym}}_{TK}(\langle \mathsf{aIndex}, j, h(A) \rangle)$ - without TK, its infeasible to determine previous anchor containing $bp(w_j)$
22. $\quad\quad \mathsf{bp}(w_j) \leftarrow \mathcal{E}^{\mathsf{ibe}}_{w_j}(C)$ - the *next* record containing w_j
23. $\quad C \leftarrow \mathcal{E}^{\mathsf{sym}}_{TK}(A)$ will be pointed to by $\mathsf{bp}(w_j)$
24. $\quad \mathsf{anchor}(\mathsf{aIndex}) \leftarrow \langle h(TK), C, \mathcal{S}^{\mathsf{sig}}_{AK}(C) \rangle$
25. $\quad \mathsf{aIndex} \leftarrow \mathsf{aIndex} + 1$

Fig. 1. The LogIt and AnchorIt algorithms

contains aIndex, backpointers for all words w provided to some LogIt function call since the last AnchorIt call (as indicated by current(w), see line 11), and a hash of the value A constructed in the last call to AnchorIt; this hash value is denoted prevAncHash(lines 17–18).

Like LogIt, AnchorIt also updates the backpointers for those words for which backpointers were included in A (lines 21–22). The primary difference in how AnchorIt creates the new backpointer for word w is that the contents are en-

crypted under *both* the time key TK (line 21) and w (line 22). As such, the contents are useful only to those who can decrypt both. As in LogIt, these new backpointers are saved for use in the next invocation of LogIt (or AnchorIt). Finally, A is encrypted under TK, and is stored signed by AK and with $h(TK)$ as its search key. Intuitively, an investigator given TK and the public key corresponding to AK can request this anchor record (by requesting $h(TK)$, verify the digital signature in the third component, and decrypt the contents (second component) to obtain A.

4.2 Authorization

Suppose an investigator requests to search for records that occurred during time intervals $T = [p_1, p_k]$ that are related to any keywords in W, i.e., by invoking $\mathsf{Auth}(T, W)$. The investigator sends the values T and W to Alice. If Alice does not approve authorization of the requested search, denied is returned. Otherwise, she provides several values to the investigator to facilitate the search (i.e., so the investigator can perform $\mathsf{Access}(T, W)$) and returns granted.

1. For each keyword $w \in W$, Alice computes $d_w \leftarrow \mathcal{X}^{\mathsf{ibe}}_{K^{\mathsf{ibe}}}(w)$ and sends d_w to the investigator.
2. Alice computes the value $AK \in \mathsf{Priv}^{\mathsf{sig}}$ provided to the first call to AnchorIt at the end of period p_k. In addition, Alice computes every $TK \in \mathcal{K}^{\mathsf{sym}}$ provided to any AnchorIt call at the end of time intervals $[p_1, p_k]$. Alice then sends these time keys and the single public verification key corresponding to AK to the investigator.

4.3 Access

For ease of exposition, we discuss the investigator's access protocol in terms of a single keyword w; this protocol can be run in parallel for multiple words. Recall from Section 4.2 that the investigator is in possession of (i) all time keys $TK \in \mathcal{K}^{\mathsf{sym}}$ that were provided to any AnchorIt invocation at the end of a period in $T = [p_1, p_k]$ – we denote these timekeys by $TK_1, \ldots, TK_k$ – and (ii) the public verification key $VK \in \mathsf{Pub}^{\mathsf{sig}}$ corresponding to the private signature key $AK \in \mathsf{Priv}^{\mathsf{sig}}$ provided to the AnchorIt call at the end of period p_k. Though Access is described in Section 2 as taking T and the set W (which in this case is $\{w\}$) as arguments, here we abuse notation and specify it taking $VK \in \mathsf{Pub}^{\mathsf{sig}}$ and $TK_1, \ldots, TK_k \in \mathcal{K}^{\mathsf{sym}}$.

Pseudocode for the Access operation is shown in Figure 2. In this pseudocode, we presume that the server provides interfaces by which the investigator can specify a value $r \in \mathsf{Range}(h)$ and request from the server the element of record of the form $\langle r, C \rangle$, if one exists, or the element of anchor of the form $\langle r, C, \sigma \rangle$, if one exists. We denote these operations by $C \leftarrow \mathsf{record.retrieve}(r)$ and $\langle C, \sigma \rangle \leftarrow \mathsf{anchor.retrieve}(r)$, respectively (lines 8, 18, 27). In addition, it is intended in Figure 2 that variables i, j, n are elements of $\mathbb{N}$; v and annotations thereof (e.g., v') are elements of $\mathsf{Range}(h)$; b and annotations thereof (e.g., b_j) are elements

Finds which of $b_1, \ldots, b_n$, if any, successfully decrypts using d_w and TK_p

1. $\mathsf{scanAnchor}(p, v, b_1, \ldots, b_n)$
2. $\quad \ell \leftarrow \arg\min_{\ell'} : \ell' > n \ \vee \ \langle v', j' \rangle \leftarrow \mathcal{D}^{\mathsf{sym}}_{TK_p}(\mathcal{D}^{\mathsf{ibe}}_{d_w}(b_{\ell'}))$ - b_ℓ properly decrypts or $\ell > n$
3. $\quad$ **if** $(\ell \leq n)$
4. $\quad\quad \mathsf{getRecord}(p, v', j')$
5. $\quad$ **else if** $(p > 1)$
6. $\quad\quad \mathsf{getAnchor}(p - 1, v, \perp)$

Retrieves an anchor, authenticates it using v, and decrypts the backpointer

7. $\mathsf{getAnchor}(p, v, j)$
8. $\quad \langle C, \sigma \rangle \leftarrow \mathsf{anchor.retrieve}(h(TK_p))$
9. $\quad A \leftarrow \mathcal{D}^{\mathsf{sym}}_{TK_p}(C)$
10. $\quad$ **assert**$(h(A) = v \ \wedge \ \langle i, v', b_1, \ldots, b_n \rangle \leftarrow A)$ - authenticate record
11. $\quad$ **if** $(j \neq \perp)$
12. $\quad\quad$ **assert**$(\langle v, j \rangle \leftarrow \mathcal{D}^{\mathsf{sym}}_{TK_p}(\mathcal{D}^{\mathsf{ibe}}_{d_w}(b_j)))$
13. $\quad\quad \mathsf{getRecord}(p, v, j)$
14. $\quad$ **else**
15. $\quad\quad \mathsf{scanAnchor}(p, v', b_1, \ldots, b_n)$

Retrieves a record, authenticates it using v, and decrypts the backpointer

16. $\mathsf{getRecord}(p, v, j)$
17. $\quad$ **repeat**
18. $\quad\quad C \leftarrow \mathsf{record.retrieve}(h(v))$ - v should be $h(R)$ of Fig.1:ln 13
19. $\quad\quad R \leftarrow \mathcal{D}^{\mathsf{sym}}_{v}(C)$ - decrypt record
20. $\quad\quad$ **assert**$(h(R) = v \ \wedge \ \langle i, m, b_1, \ldots, b_n \rangle \leftarrow R)$ - hash authenticates record
21. $\quad\quad \mathsf{results} \leftarrow \mathsf{results} \cup \{m\}$ - accumulate results
22. $\quad\quad X \leftarrow \mathcal{D}^{\mathsf{ibe}}_{d_w}(b_j)$
23. $\quad$ **while** $(\langle v, j \rangle \leftarrow X)$
24. $\quad p \leftarrow \arg\max_{p'} : p' < p \ \wedge \ (p' = 0 \ \vee \ \langle i, j, v \rangle \leftarrow \mathcal{D}^{\mathsf{sym}}_{TK_{p'}}(X))$
25. $\quad$ **if** $(p > 0)$
26. $\quad\quad \mathsf{getAnchor}(p, v, j)$

27. $\langle C, \sigma \rangle \leftarrow \mathsf{anchor.retrieve}(h(TK_k))$ - Access operation starts here
28. $A \leftarrow \mathcal{D}^{\mathsf{sym}}_{TK_k}(C)$
29. **assert**$(\mathcal{V}^{\mathsf{sig}}_{VK}(C, \sigma) = 1 \ \wedge \ \langle i, v, b_1, \ldots, b_n \rangle \leftarrow A)$ - v should be $\mathsf{prevAncHash}$ from Fig.1:ln 17
30. $\mathsf{scanAnchor}(k, v, b_1, \ldots, b_n)$

Fig. 2. The $\mathsf{Access}(VK : \mathsf{Pub}^{\mathsf{sig}}, TK_1, \ldots, TK_k : \mathcal{K}^{\mathsf{sym}})$ algorithm

of $\mathcal{C}^{\mathsf{ibe}}$; C is a member of $\mathcal{C}^{\mathsf{sym}}$; σ is a members of Σ^{sig}, and m is an element of $\{0,1\}^*$. We use this typing information, in particular, in attributing a truth value to an assignment: e.g., $\langle v, j \rangle \leftarrow X$ (line 23) is true if X is a strongly typed representation of a tuple in $\mathsf{Range}(h) \times \mathbb{N}$, in which case these values are assigned to v and j, and is false otherwise (and the assignment has no effect). The truth value of assignments is tested in both branching statements (e.g., line 23) and in **assert** statements (line 10). An **assert**(E) statement for some expression E

aborts the Access operation immediately with a return value of tampered if E evaluates to false, and otherwise simply evaluates the expression E.

The Access operation starts by retrieving the anchor record associated with $h(TK_k)$ (line 27). After decrypting it and authenticating its contents using VK (line 29), scanAnchor, is called to find if any backpointer in the anchor record properly decrypts using d_w and then TK_p (where $p = k$ in this case). Line 2 in scanAnchor deserves explanation: this assigns to ℓ the smallest value ℓ' such that either backpointer $b_{\ell'}$ properly decrypts to an element of $\mathsf{Range}(h) \times \mathbb{N}$, in which case these values are assigned to $\langle v', j' \rangle$, or $\ell' > n$. If no backpointer properly decrypts (and so $\ell' > n$), then the preceding anchor record is retrieved (line 6); otherwise the record to which the backpointer "points" to is retrieved.

The getAnchor and getRecord functions are shown in lines 7–15 and 16–26, respectively; both are straightforward in light of the log construction in Figure 1. Briefly, getAnchor is provided an authenticator v for the anchor to be retrieved, the index p for the key TK_p under which it should decrypt, and optionally an index j for the backpointer b_j it contains that is encrypted under w (and TK_p). So, getRecord retrieves the anchor record using $h(TK_p)$ (line 8), decrypts it using TK_p (line 9) and authenticates it using v (line 10), and then either calls scanAnchor to scan the record or, if an index j is provided, then b_j is decrypted and getRecord invoked.

Similarly, getRecord accepts an authenticator and decryption key v for a log record to be retrieved, and an index j for the backpointer it contains that is encrypted under w. This function loops to repeatedly retrieve (line 18), decrypt (line 19), and authenticate (line 20) a record. The contents m are added to a results set (line 21) and then b_j is decrypted. If this decryption is an element of $\mathsf{Range}(h) \times \mathbb{N}$, then this backpointer points to another log record, and in that case, the loop is re-entered. Otherwise, the backpointer points to an anchor record and the backpointer is decrypted with $TK_{p'}$ (line 24) and the getAnchor is called to retrieve the corresponding record.

The recursive calls that comprise the Access function cease when the value p reaches $p = 1$; see lines 5 and 25. At this point, all calls return and results is returned from Access.

5 Security

In this section we informally justify the claim that these protocols implement Requirements 1–4. Proofs will be provided in the full version of this paper.

Liveness. Recall that Access returns tampered only when the expression E in an **assert**(E) evaluates to false. Liveness can be confirmed through inspection of Figures 1 and 2 to determine that if $\mathsf{Access}(T, W)$ is executed by an honest investigator before the machine is compromised, then it will not return tampered.

Soundness. Soundness requires, intuitively, that an attacker who compromises the server is unable to undetectably insert new records into the log, or alter

records in the log, for any time period in $T = [p_1, p_k]$ (i.e., the searched interval), where time period p_k is completed by the time the compromise occurs. Recall that p_k is terminated with a call to AnchorIt, where an authentication key AK for time period p_k is used to digitally sign the anchor record (Figure 1, line 24). At that point, AK should be deleted from the server (not shown in Figure 1), and so will not be available to the attacker after compromising the server. Since the investigator is provided the public key VK corresponding to AK with which to authenticate the first record she retrieves (see Figure 2, line 29), the attacker is unable to forge this record. Moreover, every log or anchor record retrieved provides a value $v \in \mathsf{Range}(h)$ with which the next record retrieved is authenticated; see Figure 2, lines 10, 20. Via this chaining, the attacker is precluded from inserting log records undetectably (with overwhelming probability).

Completeness. Completeness requires that an attacker who compromises the server is unable to undetectably delete records from the log for any time period in $T = [p_1, p_k]$, where time period p_k is already completed by the time the compromise occurs. This reasoning proceeds similarly to that for soundness, noting that each record retrieved is authenticated using information from the previous record. This chaining ensures that any gaps in the chain will be detected (with overwhelming probability).

Privacy. Privacy is the most subtle of the properties. Informally, we are required to show that the only records an investigator is able to retrieve from a non-compromised server are $\mathsf{match}(\cup_i T_i, \cup_i W_i)$. We assume here that the only means the investigator has for retrieving records is the anchor.retrieve and record.retrieve interfaces utilized in Figure 2.

Informally, in order to retrieve an anchor record, it is necessary for the investigator to know $h(TK)$ for the corresponding time key TK. This can be obtained only by obtaining TK directly from Alice; time keys or hashes thereof appear nowhere in the contents of log records (except as encryption keys, but since IND-CCA security implies security against key recovery attacks, the time key or its hash is not leaked by the ciphertexts). As such, if the investigator can retrieve $h(TK)$ from the server, then $TK \in \cup_i T_i$. Similarly, to retrieve a log record, the investigator must know $h(h(R))$ for the log record R. In this case, the value $h(R)$ does appear in log records, specifically in backpointers, but always appears encrypted by a word w that characterizes it (Figure 1, line 12). These backpointers, in turn, are included in log records or anchor records (Figure 1, lines 9, 17), and by a simple inductive argument, it is possible to argue (informally) that any log record that can be retrieved requires knowledge of both d_w for a word w that characterizes it, and the time key TK for the time period in which it was written.

A formal proof of this property requires substantially greater care and rigor, of course. In particular, it requires us to model h as a random oracle [2], since its outputs are used as encryption keys for records (Figure 1, line 13). In addition, since h returns a unique value per input, it is essential that no two records R

be the same, though this is assured since each record R includes backpointers, themselves ciphertexts generated by a randomized encryption algorithm.

We note that our requirements in Section 2 impose no limits on the information leaked to the adversary who compromises the server, except where this information might permit the attacker to compromise Soundness or Completeness. That said, certain steps in our protocols (e.g., encrypting the records containing message m) are taken for privacy versus this adversary, and indeed our protocol successfully hides information about log records written in time period *before* that in which the compromise occurs. Records written before the compromise, but within the same time period in which the compromise occurs, are not afforded the same protection.

6 Experimental Evaluation

In this section we evaluate the performance of our proposed technique for searching encrypted audit logs. All experiments were performed on a dual 1.3 GHz G4 server with 1 GB of memory. 128-bit AES was used for symmetric operations, 160-bit SHA-1 as our hash function, and DSA [8] with a 1024 bit modulus and 160 bit secrets for the digital signature operations. A 512-bit prime and a subgroup of size 160 bits was used for IBE operations [14].

Performance was evaluated by replaying Snort [6] IDS events recorded on a server over a period spanning several months. The events in the log occur in bursts, with half occurring in a 10 day span. During the observed period, approximately 27,000 alerts were recorded and includes attacks from nearly 1700 distinct IP addresses, with roughly 500 of these IPs involved in more than 10 alerts. On average, records were described by 6 keywords and the encrypted log required 62.5 MB of storage.

Unfortunately, due to its bursty nature, the Snort log is not as diverse as one would expect, and not well suited for testing the average case performance. To address this, we evaluated the performance characteristics on a synthetic log with characteristics similar to that of the Snort log. The synthetic log contains 30,000 entries timed with an exponential distribution. The expected elapsed time between records was set to 15 minutes, resulting in a log that spans roughly 11 months. To closely approximate the Snort log, records are described with an average of 5 keywords that include one of 10 random attack types, one of 500 randomly chosen IP addresses, keyword ALL, and keywords that appear with known probabilities.

To evaluate the efficiency of our approach we experimented with searches on both the real and synthetic logs. These searches span the entire log, and for a fair comparison to [17], we assume that summary blocks occur once per time zone. For the synthetic log, searches are performed for keywords occurring with varying probability. The left-hand side of Figure 3 depicts the results of searches averaged over 100 randomly generated logs. The results show that when searching for keywords that appears infrequently, our technique incurs significantly less performance penalty than that of [17]. For example, if the search keyword

occurred within 5% of the records in the log, our technique results in roughly 10 times fewer IBE decryptions than the enhanced index scheme of [17]. Moreover, if the keyword being searched appears in less than 1% of the records (e.g, an IP address), our approach requires roughly forty times less operations than the enhanced scheme of [17].

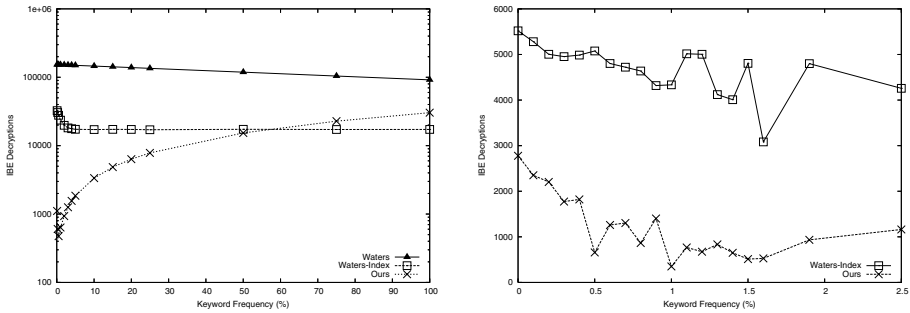

Fig. 3. Incurred IBE decryptions as a function of the keyword frequency. (Left) Searching on the synthetic logs averaged across 100 runs. (Right) Searching on the Snort log

To see why this is the case, let e denote the number of entries in the log, b the number of log entries per block, u the average number of distinct keywords in a block, and k the number of log records that are related to a keyword w. To perform a search for w using the scheme of [17] an investigator must retrieve $O(e/b + k)$ log records and perform $O(u \cdot \frac{e}{b})$ IBE decryptions. In practice, since most useful searches are for infrequently occurring keywords then $e/b \gg k$, which means that there are many distinct keywords in each block and so $u > b$. Hence, the number of records retrieved (and associated IBE operations to be performed) becomes proportional to the total number of records (e), rather than the number of *matching* records, k. By contrast, our scheme is similar in performance to that of [17] up until the first matching record is found, after which the number of retrieved records and IBE decryptions are both $O(k)$.

Similar performance improvements were observed for searches on the Snort log (shown in the right-hand side of Figure 3). There, each point depicts the average number of IBE decryptions required to separately search for each keyword that appears in the specified percent of records.

References

1. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. In *Proceedings of the* 38^{th} *Symposium on Foundations of Computer Science*, 1997.
2. M. Bellare and P. Rogaway. Random oracles are practical: A Paradigm for Designing Efficient Protocols. In 1^{st} *ACM Conference on Computer and Communications Security*, pages 62–73, November 1993.

3. S. M. Bellovin and W. R. Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters. Cryptology ePrint Archive, Report 2004/022, 2004.
4. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. Cryptology ePrint Archive, Report 2003/195, 2004.
5. D. Boneh and M. Franklin. Identity Based Encryption from the Weil Paring. In proceedings of CRYPTO '2001, LNCS, Vol. 2139, pp. 213-229, 2001.
6. B. Caswell and J. Beale and J. Foster and J. Faircloth. Snort 2.0 Intrusion detection system. May, 2004. See http://www.snort.org
7. Y. Chang and M. Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. Cryptology ePrint Archive, Report 2004/051, 2004.
8. Federal Information Processing Standards. *Digital Signature Standards (DSS) – FIPS 186*, May, 1994.
9. E. Goh. Secure Indexes. Cryptology EPrint Archive, Report 2003/216, 2003.
10. S. Goldwasser, S. Micali, and R. L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal of Computing* 17(2):281–308, April 1988.
11. S. Harber and W. Stornetta. How to Time-Stamp a Digital Document. In A. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, LNCS vol 537, pages 437-455, 1991.
12. J. Kelsey, and B. Schneier. Minimizing Bandwidth for Remote Access to Cryptographically Protected Audit Logs. In *Web Proceedings of the* 2^{nd} *International Workshop on Recent Advances in Intrusion Detection*, 1999.
13. M. Mont, K. Harrison, and M. Sadler. The HP Time Vault Service: Exploiting IBE for Timed Release of Confidential Information. In Proceedings 13^{th} *Annual WWW Conference, Security and Privacy Track*, 2003.
14. Stanford Applied Cryptography Group. IBE Secure Email. See http://crypto.stanford.edu/ibe.
15. B. Schneier, and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the* 7^{th} *USENIX Security Symposium* pp. 53-62, 1998.
16. D. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In Proceedings of *IEEE Symposium on Security and Privacy*, May 2000.
17. B. R. Waters, D. Balfanz, G. Durfe, and D. K. Smetters. Building an Encrypted and Searchable Audit Log. In *Proceedings of Network and Distributed System Symposium*, 2004.