Building an Encrypted and Searchable Audit Log

Brent R. Waters[1*], Dirk Balfanz[2], Glenn Durfee[2], and D. K. Smetters[2]

[1] Princeton University
Computer Science Department
Princeton, NJ 08544
bwaters@cs.princeton.edu

[2] Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
{balfanz, gdurfee,
smetters}@parc.com

Abstract

Audit logs are an important part of any secure system, and they need to be carefully designed in order to give a faithful representation of past system activity. This is especially true in the presence of adversaries who might want to tamper with the audit logs. While it is important that auditors can inspect audit logs to assess past system activity, the content of an audit log may contain sensitive information, and should therefore be protected from unauthorized parties.

*Protecting the contents of audit logs from unauthorized parties (*i.e., *encrypting it), while making it efficiently searchable by authorized auditors poses a problem. We describe an approach for constructing searchable encrypted audit logs which can be combined with any number of existing approaches for creating tamper-resistant logs. In particular, we implemented an audit log for database queries that uses hash chains for integrity protection and identity-based encryption with extracted keywords to enable searching on the encrypted log. Our technique for keyword search on encrypted data has wide application beyond searchable audit logs.*

1. Introduction

System logs provide an invaluable view into the current and past state of almost any type of complex system. Most server software in existence today includes some logging mechanisms.

Secure versions of such logs, designed to defend against malicious tampering, allow the current state of the system to be audited even when that system has been under active attack by malicious insiders or outsiders [7, 9]. Correctly designed secure audit logging mechanisms can detect unauthorized past activity, even when the person performing that action goes to great lengths to cover their tracks. The existence of such logs can be used to enforce correct user behavior, by holding users accountable for their actions as recorded in the audit log. Such logs can be used in a wide variety of systems, from a control system that logs the commands a user issues, to a database system that logs the queries a user makes.

Typically, when an organization wishes to inspect past activity it will search the audit log for relevant information. For example, if a certain user was suspected of behaving improperly the organization might search for all actions performed by that particular user. If the organization wishes to see all actions of a certain type, it might search for all log entries that match a given keyword. For an audit log to be useful in practice, it is critical that it be efficiently searchable for keywords of interest.

At the same time, the contents of an audit log can be considered to be sensitive information. For instance, knowing what actions are made by a certain user could violate that individual's privacy. If the log contains information about not only what query was made, but what results were returned, access to the audit log would imply effective access to the database, circumventing database access controls. The organization that owns the system being logged might consider the information the log holds to be valuable and not wish to share it with others, while for robustness' sake, the organization may want to store backup copies of the audit log information at sites it may not completely control. In general, this means that the contents of the audit log must be encrypted. However, this makes it extremely difficult to search.

Using traditional techniques, searching the log would re-

*The majority of this work was completed while the author was a summer intern at PARC.

quire decrypting every record. This approach has several disadvantages. First, it requires decrypting all of the log data, regardless of what information one is looking for; this opens opportunities for unintended access to log records other than the ones relevant to the current investigation. Second, it requires the entity with the decryption key to interactively process all the log data, which can be quite large. In many applications, one would like to entrust the ability to decrypt audit logs to an entity or system with high levels of trust and assurance; requiring that system to also be able to process large quantities of log data in an on-line fashion limits one's choice of trusted parties. It would be preferable to be able to selectively delegate the ability to search the log to parties with the means to process the data.

The key challenge to building a successful, secure audit logging system is to simultaneously protect the integrity of the audit log, control access to contents, and maintain its usefulness by making it searchable.

In this paper, we present a design for an encrypted audit log that allows a designated trusted party, the *audit escrow agent*, to construct *keyword search capabilities*, which allow (less trusted) *investigators* in possession of such capabilities to search for and decrypt entries matching a given keyword. The escrow agent can distribute a capability to an investigator if he deems it appropriate. Since we expect keyword search capabilities to be distributed rather infrequently, the escrow agent can be made to be very secure from attack.

We developed a public key based cryptographic scheme that allows keyword searching on encrypted data by adapting Boneh and Franklin's [3] Identity-Based Encryption (IBE) scheme. (We note that the cryptographic scheme we use is similar to a scheme that was independently discovered by Boneh et al. [2]; see Section 2 for details.) In an IBE scheme, public keys can be arbitrary strings – *e.g.*, "bob@parc.com". Private keys are derived from public keys through use of a system-wide *master secret*, known by a trusted authority. In our design, search keywords are used as IBE public keys, and the *master secret* is held by an authority trusted to issue keyword search capabilities for a given audit log, in our case, the *audit escrow agent* described above.

In our design, the server generating audit log entries encrypts entries with the public keys corresponding to the keywords that are derived from those entries. The escrow agent, which holds the IBE master secret, can construct a search capability for a given keyword as the private key corresponding to the given keyword. Furthermore, additional security properties of Boneh and Franklin's scheme imply that an adversary cannot tell which public key was used to create a ciphertext when given the ciphertext. Thus, when an encrypted audit log entry is created, even its search keywords are hidden.

The rest of the paper is organized as follows. In Section 2 we describe related work. Sections 3 and 4 introduce secure audit logs in general, and our system in particular. Section 5.1 presents a symmetric key based scheme, while Section 5.2 presents an public-key scheme based on IBE. In Section 6 we present our implementation of a proxy server that creates a searchable audit log of database queries and discuss its performance. Finally, we conclude in Section 7.

2. Related Work

SEARCHING ON ENCRYPTED DATA. Song et al. [11] study the problem of searching on encrypted data in a symmetric-key setting. In a symmetric key based scheme, the keys that are used to create the encrypted entries also allow search and decryption of the audit log. Thus, servers that construct audit log entries possess keys capable of decrypting log entries. We discuss the shortcomings of such an approach in Section 5.1 and contrast it with a public-key based scheme. Our public-key based scheme easily allows audit escrow agents (and only those agents) to create capabilities to search the audit log for certain keywords.

Goh examines how Bloom Filters can be used to make searching on encrypted data more efficient [4]. Like Song et al., Goh presents a scheme in the symmetric key setting. He presents a scheme where a encrypted data consists of the encryption of a document and a Bloom Filter attached that is used for keyword searching.

Boneh et al. [2] have also recently examined the problem of searching on publicly encrypted data. They independently devised a scheme based on the Identity-Based Encryption scheme of Boneh and Franklin [3]. Their scheme is similar to our underlying cryptographic scheme in its construction and security properties. The contribution of their work is different, however: they provide a detailed theoretical analysis which includes a precise definition of what they call *searchable public-key encryption*, along with three constructions that are provably secure in their model under suitable cryptographic assumptions. Our work, on the other hand, introduces our independently developed construction and focuses on the pragmatic security concerns regarding integrating it in a system for creating secure audit logs.

AUDIT LOGS. Schneier and Kelsey [7, 8, 9] describe a secure audit logging scheme capable of detecting any attempt to delete or alter past audit log entries, even on a host that has been compromised (assuming the entries were made before the compromise). Such tampering can be detected even if the compromised host has not been able to offload any state information to another host; an operation referred to as "checkpointing" in the discussion below. To accomplish this, a system opening a new audit log first establishes a shared secret A_0 with a trusted third party. After

each audit record is generated, the current shared secret, A_i, is *evolved* – it is completely replaced by a new shared secret, A_{i+1}, computed as the cryptographic digest of the previous shared secret, A_i. Each audit record is encrypted under a key K_i which is derived from the current value of A_i, and then the encrypted record is protected using a Message Authentication Code (MAC) keyed with A_i. Records are linked using a hash chain [6].

Because the secrets used to encrypt and authenticate each log record are completely replaced on the logging host after the record is generated, an attacker compromising that host does not have the necessary information to go back and replace, delete, or modify existing log records stored on that host. Any attempt to do so can be detected by the trusted third party, who retains A_0,and can check that there is a valid record authenticated with each MAC key A_i. This constitutes a form of *forward security* for the audit log.

The use of symmetric MACs to authenticate log records means that only the trusted third party (or someone to whom it has delegated a record authentication key, A_i) can verify the audit log. As each record is encrypted with a different key, the trusted third party can delegate the ability to decrypt particular audit records to designated individuals, by giving them the keys used to encrypt those records. However, it does not allow any form of search on the encrypted audit data.

3. Characteristics of a Secure Audit Log

We can identify three important properties a secure audit log: those designed to prevent and detect tampering, and those designed to control data and search access.

TAMPER RESISTANCE. A secure audit log must be *tamper resistant* – it must guarantee that no one other than the creator of the log can create valid entries, and that once entries have been created, they cannot be altered.

One cannot prevent an attacker who has compromised the system creating the log from altering what that system will put in future log entries [7]. One also cannot prevent him from deleting any log entries that have not already been copied to another system. The goal of a secure audit log in such cases is to make sure that he cannot alter existing log entries, and that any attempts to delete such existing entries will be detected. Ideally, one would like to detect attempts to delete or alter any entries created up to the time a host is compromised [7, 8]. For for some applications it may be enough to have the logging host "checkpoint" its state periodically – to copy its log data, or some function (*e.g.,* a signature) of its log data to another host, and simply be able to assure that no entries up till the most recent checkpoint have been deleted or altered.

VERIFIABILITY. A secure audit log must also be *verifiable* – it must be possible to check that all entries in the log are present and have not been altered. Audit logs can either be *publicly verifiable* – verifiable by anyone holding appropriately authenticated public information, *e.g.,* the logging system's public key, or an authenticated hash of all existing audit entries. Or, they may require a *trusted verifier* – they can only be verified by a designated party holding one or more secrets, *e.g.,* a MAC key. The choice of approach is application-dependent. Publicly verifiable audit log systems, *e.g.,* systems that simply digitally sign each log entry they generate, allow easy storage of audit logs on untrusted systems, and the increased trust resulting from the ability of any interested party to verify the log. On the other hand, trusted verifier systems, such as the Schneier and Kelsey scheme described in [7, 8, 9] allow for a greater degree of forward security in an audit log system, making it possible to detect attempts to delete audit log entries made any time before a system is compromised, without requiring any information about those entries to be communicated to the outside world.

To verify an audit log, it must contain two types of information. First, each entry must contain enough information to verify its authenticity when considered on its own. If some entries are altered or deleted, the ability to individually verify the remaining entries (or blocks of entries) makes it possible to recover some useful information from the damaged log. Second, the individual entries must also be linked together in a way that makes it possible to determine whether any entries are missing. Serial numbers allow one to check whether all entries are present, but turn the problem of tampering with the log into one of attacking each entry individually. Hash chaining [6, 7, 8, 9], where each entry contains a cryptographic digest of the previous entry, is a better solution, as it tightly links all entries in the chain.[1] It also allows a very simple form of public verifiability, where the hash of the most recent audit entry is *checkpointed, i.e.,* published via a trusted third party (*e.g.,* the New York Times).

DATA ACCESS CONTROL AND SEARCHABILITY. Given that the data in an audit log may be sensitive, it must be encrypted. However, one would like to be able to allow legitimate search access to a subset of all audit log entries (*e.g.,* all entries matching the keyword "Smith"). We present a new criteria for the construction of useful secure audit logs, namely that they allow the secure delegation of search capabilities.

Delegation of capabilities is important so that an investigator can search and view entries of a narrow scope. For example, if Alice Smith wanted to investigate all entries related to her the audit escrow agent might give her the capability to search for all entries matching the keyword

[1] In order to provide individual entry verifiability in a hash-chained audit log, each log entry must explicitly contain the hash of the previous entry to allow some recovery if that entry is missing.

"Smith", but not give her anything more. The alternative of having the master secret holder perform the searches is undesirable since it unnecessarily exposes a highly trusted component of the system.

For such delegation to be considered secure, it must be impossible for an adversary to learn the content of entries in the audit log that he should not have access to (up to the security provided by the underlying encryption function, which might not, for instance, disguise characteristics such as the length of the audit log entry.) We allow our adversary to be an insider in the sense that he may be both a user of the system, and may have had some legitimate search capabilities explicitly given to him by the audit escrow agent. We would like to ensure that, assuming he does not compromise the escrow agent itself, he is unable to view the contents of any audit log entry, or even to learn which keywords match an entry beyond those set of keywords and entries for which he has legitimate access.

4. Audit Log Components and Notation

To make our presentation concrete, we take as an example the problem of logging queries made by a set of authenticated users against one or more SQL databases. The mechanisms we describe can also be applied directly to generate searchable secure audit logs for other system types – only the actual content to be logged, and the choice of keywords to support for search on that content need to be customized to the application or system to log.

Our audit log L consists of a series of individual *audit records*, $R_0, R_1, \ldots, R_n$. Each record R_i contains:

1. $E_{K_i}(m_i)$, the encryption of the data to be logged under a key K_i. The string m_i consists of the database query to be logged, along with metadata such as the identity of the user who issued the query. Optionally, it could also contain the query results. In our system, the key K_i is chosen randomly for each log entry.
2. $H(R_{i-1})$, the hash of the previous record, to form a hash chain.
3. $c_{w_a}, c_{w_b}, c_{w_c}, \ldots$, information about the keywords $w_a, w_b, w_c, \ldots$ that can be used for searching.
4. Verification information V_i. In our implementation, this is simply the hash to date of the current chain of audit records (*i.e.,* $H(R_i)$). We note that we could also use a standard public key signature, or a MAC created using a key shared with a trusted verifier. If that shared key evolves with each record, we get desirable forward security properties as described in [7, 8]). V_i must authenticate all of the other data in the audit record, including the keyword information c_{w_n}.

To construct a searchable secure audit record R_i, the server first extracts keywords that characterize the record. These are the keywords that can be used to search for that record in the future. Next, it encrypts the entry using the key K_i, producing the keyword search information c_{w_n} in the process. In Sections 5.1 and 5.2 we present two concrete instantiations of this procedure. Finally, the server constructs the verification data V_i. In our implementation, we periodically "checkpoint" the audit log by publishing the most recent verification value, V_i, to one or more other servers, producing a publicly verifiable audit log.

KEYWORD EXTRACTION. In our implementation, queries are made in SQL. See Figure 1 for an explanation of how we extract keywords from a query. Our set of keywords not only contains keywords from the query, but also metadata such as the user who made the query and the time when the query was issued. Note that we prefix keywords with suitable labels so that we can distinguish the case where user "Alice Smith" is making a query from the case where someone makes a query mentioning the name "Alice Smith".

5. Searching on Encrypted Queries

If at some point an investigator wants to search an audit log for entries matching a certain keyword, she must go to the audit escrow agent for the organization that generated the log and request a search capability for that keyword. If the escrow agent deems it appropriate, he grants this capability to the investigator. She may then go to the audit log and search through the entries and see which entries match the keyword. For those audit log entries that match the keyword, the investigator can decrypt the entry and view its contents (see Figure 2).

In this section we present two schemes for creating encrypted and searchable entries. We first present a scheme based on symmetric key cryptography. Although this scheme is secure against a passive adversary, we find that the scheme is insecure against an adversary that is able to compromise an audit log server. The second scheme we present is based on asymmetric key cryptography and addresses this issue.

5.1. Symmetric Key Scheme

We describe a symmetric key based scheme for encrypting searchable audit log entries. Our method is derived from previous work on searching on encrypted data [4, 11].

Setup: Suppose there are t audit log servers. The audit escrow agent generates independent and uniformly random secrets $S_1, \ldots, S_t$ and gives S_j to the jth server.

Encryption: Suppose the audit escrow agent has issued a secret S to a particular audit log server. Let H be a keyed pseudorandom function (PRF); we denote by H_S the PRF

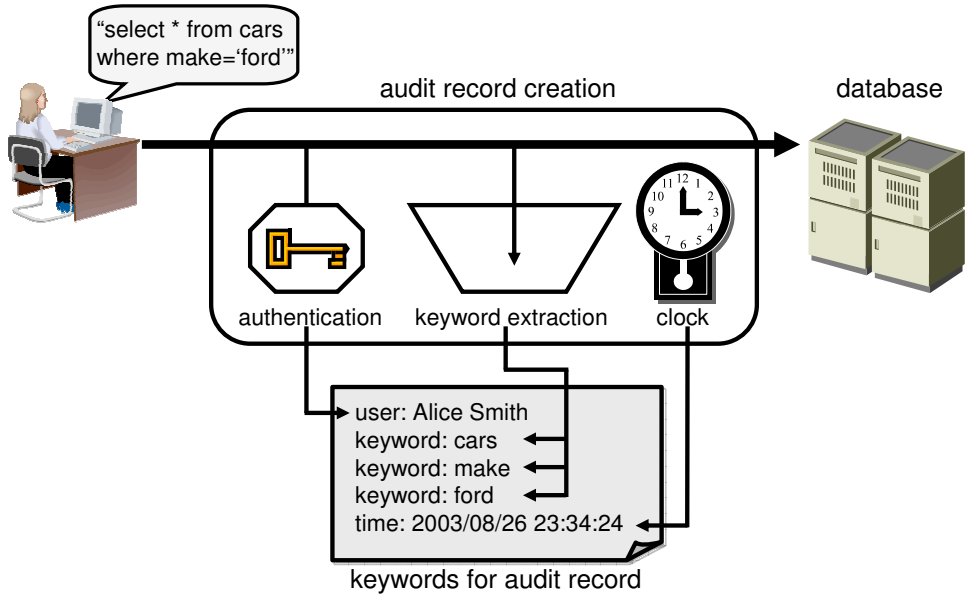

Figure 1. Extracting keywords for an audit record: audit records cannot only be searched by keywords contained in the query logged, but also by meta-data such as user name and time.

H keyed with the secret S. In practice, HMAC-SHA1 can be used in place of H. Let E be a symmetric encryption function; we denote by E_K the function E keyed with K.

Suppose the server is to encrypt the log entry, m, along with keywords $w_1, w_2, \ldots, w_n$. Let **flag** be a constant bit-string of length ℓ. The server executes the following steps:

1. The server chooses a random symmetric encryption key, K, to be used only for this entry.
2. The server computes the encryption $E_K(m)$.
3. The server chooses a random bit string r of some fixed length. The random string r is uniformly independently drawn for each entry.
4. For i from 1 to n the server computes

$$a_i := H_S(w_i), \quad b_i := H_{a_i}(r), \quad c_i := b_i \oplus (\mathbf{flag}|K).$$

   In other words, for each keyword w_i the PRF is first keyed with S and is given input w_i. The result a_i is then used to key the PRF which is then called with input r to give b_i. The result b_i is then XORed with the concatenation of **flag** and the symmetric key K to give the output c_i.
5. The server writes $\langle E_K(m), r, c_1, c_2, \ldots, c_n \rangle$ as the encrypted entry to the audit log.

Informally, an adversary that does not know S is unable to compute $a_i = H_S(w_i)$, and thus, b_i, as long as the keyed PRF H is secure. The adversary is thus unable to learn K, and therefore cannot decrypt the entry. Additionally, the adversary is unable to link queries that had similar keywords, since r is an uniformly independent random value.

Search and Decryption: Recall there are t audit log servers in our scheme, with the jth server holding a secret S_j. Suppose an investigator wishes to obtain a search capability for the keyword w. The audit escrow agent (if he approves) constructs the search capability as

$$d_w := \langle H_{S_1}(w), \ldots, H_{S_t}(w) \rangle.$$

We denote $d_w^j := H_{S_j}(w)$ as the search capability component corresponding to the jth server.

Once given the capability, the investigator visits each audit log server. At the jth server, the investigator executes the following:

1. The investigator computes $p := H_{d_w^j}(r)$, where r is the random string stored with the query.
2. For each c_i in the entry, the investigator computes $p \oplus c_i$. If the first ℓ bits of the result matches **flag**, then the party extracts K as the remainder of the result; otherwise, the computation is disregarded. If none of the results begin with **flag**, then the query is not a keyword match, and the investigator moves to the next query.
3. If one of the results did match, the investigator uses the computed K to decrypt $E_K(m)$ to obtain m, the original audit log entry.

Suppose when encrypting an entry, the jth server uses the keyword w_i to create c_i. If $w_i = w$, we have $H_{d_w^j}(r) \oplus c_i =$

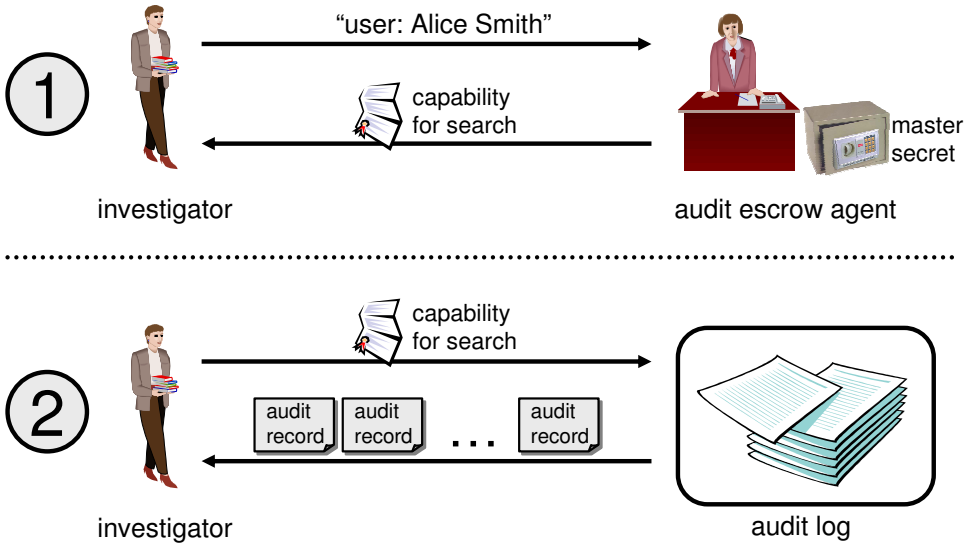

Figure 2. Searching the log: First, the investigator has to obtain a search capability for the keyword in question. Then, the she can search the audit log for that keyword.

(**flag**$|K$); otherwise, this XOR will look random. Therefore, the beginning bits of the result can be tested against **flag** to determine if there is a match. (There is a 2^{-l} chance of a false positive from this check. However, even in the event of a false positive from this check the decryption attempt will fail with very high probability. Since the length of ℓ does not actually affect the security of the scheme, the length of the bitstring **flag** can have a length significantly less than that of an encryption key. We note that a CRC or other simple checksum could also be used.) In the case of a match, the remainder of the result is the symmetric key K, which can be used to decrypt the query.

We note that the use of a pseudorandom function H to derive the search keyword capability implies that capabilities for different keywords appear to be independently random. In other words, an investigator receiving a capability d_w for a keyword w learns no new information about the capability corresponding to any other keyword w'.

Discussion: The primary problem with the symmetric method occurs in the case where an adversary is able to compromise a server's secrets. If the adversary learns S_j, he can be able to create a search capability for any keyword that he wishes that can be used to search and decrypt on the jth server.

This problem is partially alleviated if we allow the server keys to be updated or evolved over time. If a particular secret S_j was stolen from a server that used a key-update scheme then the adversary will be able to use S_j to search all entries that were created since the last update, however, he would not be able to read past log entries.

Nevertheless, even with an efficient key update mechanism, this scheme has serious drawbacks. The most significant is that in order for the servers to be updated, the audit escrow agent must have a "live" connection to a network shared with the servers. This makes the audit escrow agent more vulnerable to attacks. Another concern is that after v key updates for each of t servers, the size of a keyword search capability is proportional to vt (which can become quite large). Finally, if the adversary compromises the server, he may be able to learn a secret that would allow him to act as the server and receive key updates from the audit escrow agent. In this event, the audit log entries from a compromised server will continue to be vulnerable, even after the compromise was detected and the server repaired. These security issues indicate that it is best to put as little secret information into a server as possible, motivating the asymmetric scheme outlined in the next section.

5.2. Asymmetric Scheme

The shortcomings of the symmetric key based scheme suggest that an asymmetric key based scheme is necessary. We now present an asymmetric key based scheme for creating encrypted and searchable log entries. Our scheme is based on the Identity-Based Encryption scheme of Boneh and Franklin [3]. We first provide the reader with a brief review of IBE, then describe our scheme and discuss its attributes.

Identity-Based Encryption: In this section, we provide a brief review of Identity-Based Encryption and some nec-

essary mathematical details.[2] The Identity-Based Encryption scheme we use is based on Tate pairings over supersingular elliptic curves.

In an Identity-Based Encryption scheme, any arbitrary string can comprise a public key. If Alice wishes to send a message to Bob, she simply uses a string uniquely identifying Bob – say "bob@parc.com" – as the encryption key to encrypt her message. A system-wide master secret is used by a trusted escrow agent to generate the private key corresponding to a public key. Bob authenticates to the trusted third party (in the same way he might authenticate to a CA) to obtain the private key corresponding to "bob@parc.com", which he then may use for decryption.

IBE SETUP. To set up the system, one first selects large primes p and q, two groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order[3] q, and an arbitrary generator $P_0 \in G_1$. One also picks an admissible bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ and two cryptographic hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \mathbb{G}_2 \to \{0,1\}^n$. The master secret is a random value $s \in \mathbb{Z}_q$, known only to the trusted escrow agent. The system parameters are

$$P = (p, q, \mathbb{G}_1, \mathbb{G}_2, e, P_0, P_1), \text{ where } P_1 = sP_0,$$

and are known by all parties.

IBE KEY GENERATION. To issue the private key corresponding to the public key w, the escrow agent uses the master secret s to compute $d_w := sH_1(w) \in \mathbb{G}_1$.

IBE ENCRYPTION. To encrypt the plaintext $m \in \{0,1\}^n$ using a string w as the public key, one (1) computes $Q_w = H_1(w) \in \mathbb{G}_1$, (2) computes $g_w = e(Q_w, P_1)$, (3) picks a random $r \in \mathbb{Z}_q$, and (4) computes

$$c = \langle rP_0, m \oplus H_2(g_w^r) \rangle.$$

IBE DECRYPTION. To decrypt a ciphertext $c = \langle U, V \rangle$ using d_w as the private key, one computes

$$m = V \oplus H_2(e(d_w, U)).$$

Since e is a bilinear map[4], it follows that decryption operation is the inverse of the encryption operation. We refer the reader to [2, 3] for the details regarding the security of this scheme.

[2]There are actually two IBE schemes described by Boneh and Franklin: the simpler is semantically secure, the other satisfies chosen ciphertext security. To keep the presentation clear, we base our discussion on the semantically secure scheme. It is straightforward to generalize our work to the scheme satisfying chosen ciphertext security, and we recommend using their more secure scheme in any implementation of the secure audit log system described here.

[3]To be precise, for the scheme we use, $\mathbb{G}_1$ is an order-q subgroup of elliptic curve group over a supersingular elliptic curve, while $\mathbb{G}_2$ is an order-q subgroup $\mathbb{F}_{p^2}$.

[4]That is, $e(aP, bQ) = e(P,Q)^{ab}$ for all $P, Q \in \mathbb{G}_1$ and all $a, b \in \mathbb{Z}_q$.

Setup: To set up our scheme, we first set up an instance of the above Identity-Based Encryption scheme. In our system, the audit escrow agent is given the IBE master secret s, and all servers that contribute to the audit log are given the system parameters P.

Encryption: Suppose the server is to encrypt the log entry m, along with keywords $w_1, w_2, \ldots, w_n$. The server performs the following steps:

1. The server chooses a random symmetric encryption key, K, to be used only for this entry.
2. The server encrypts the log entry using K, to get $E_K(m)$.
3. For each keyword w_i, the server computes the Identity-Based Encryption c_i of the string $(\textbf{flag}|K)$ using w_i as the public key and P as the public parameters.
4. The server writes $E_K(m), c_1, c_2, \ldots, c_n$ as the entry to the audit log.

Since a new key K is generated for each log entry (and is thrown away by the server immediately after the log entry is generated), the only way to recover a log entry is to decrypt one of the c_i's and obtain K. It follows directly from the security of Identity-Based Encryption [3] that the only way to recover m is to know the private key corresponding to one of the keywords w_i. The particular Identity-Based Encryption scheme we have chosen also satisfies a stronger security property, namely *key-privacy* [1], which implies that an adversary can obtain no information about what public key w_i was used to produce any ciphertext c_i. (We refer the reader to [2] for a detailed proof of the key-privacy property for IBE.) This implies that the presence of the c_i in the log entry reveals no information about what keywords are present in the log entry, and an attacker cannot correlate entries in the audit log based on their keyword tags.

Search and Decryption: Suppose an investigator wishes to obtain a search capability for the keyword w. The audit escrow agent (if he approves) constructs the capability d_w as the Identity-Based Encryption private key corresponding to the string w. For each audit log entry, the investigator executes the following:

1. For each c_i the investigator attempts to IBE-decrypt c_i using the private key d_w. If the prefix of the result matches **flag** then the investigator extracts K as the remainder of the result. If none of the results begin with **flag** then the log entry does not match and the investigator moves to the next log entry.
2. If one of the results did match, the capability holder may compute K to decrypt $E_K(m)$ to obtain m.

Notice that the investigator holding some capability d_w for keyword w will not be able to gain a capability $d_{w'}$ to search for another keyword w'. Again, this follows directly from the security of the Identity-Based Encryption scheme: the capabilities correspond to different private keys in an Identity-Based Encryption scheme, which cannot be derived from each other, even if large numbers private keys are known.

Discussion: This asymmetric scheme corrects many of the drawbacks of the symmetric scheme. Since each server only stores public parameters, there are no secret keys for an attacker to steal. Compromising a server does not allow the attacker to search or decrypt any entries in the audit log that have already been generated and stored.

A drawback of this scheme is the performance overhead of using Identity-Based Encryption; however, optimizations (discussed in the next section) are available for speeding up our use of IBE.

We note that this scheme is also easy to modify to allow separating the ability to find records matching a given keyword from the ability to decrypt those records. To do this, we omit the record key K in the IBE encryptions performed that generate the tags c_i (leaving only the **flag**), and add an encryption of K encrypted under another public key belonging to the escrow agent. This introduces an extra "round trip" to the escrow agent to decrypt those records for which a match is discovered.

5.3. Optimizations for the Asymmetric Scheme

The operations in the asymmetric scheme are significantly more expensive than those of the symmetric scheme. The main bottlenecks are the computations of the pairing and modular exponentiations for each keyword w. However, if the same keywords are used frequently then intermediate results can be reused. We discuss three such optimizations in this section.

PAIRING REUSE. Our first observation is that the computation of g_w only needs to be performed once per keyword. Subsequent Identity-Based Encryptions using w as the public key can reuse g_w if it has already been computed for some other log entry. Encryption then simply becomes a matter of picking a random r and following steps (3) and (4) of encryption (see explanation of Identity-Based Encryption above). This speeds up encryption: over a set of log entries in which a keyword repeated k times, only one pairing operation and k modular exponentiations are required.

INDEXING. Further savings are possible by creating an index of keywords at periodic intervals in the log, instead of storing IBE encryptions with each log entry. If the system design allows buffering of entries sent to the audit log, then the servers may collect queries into "blocks" to be sent to the audit log all at once.

Suppose a server collects log entries $m_1, \ldots, m_t$ to be sent to the audit log, sharing in total the set of keywords $w_1, \ldots, w_u$. The server creates an audit log block and index as follows.

1. The server chooses random symmetric encryption keys, $K_1, \ldots, K_t$, for one-time use.
2. The server encrypts each log entry m_i using K_i, to get $E_{K_i}(m_i)$.
3. For each distinct keyword w_j, the server finds the indices $\{i_{j,1}, \ldots, i_{j,\ell(j)}\}$ for which w_j is a keyword where $\ell(j)$ is the number of entries for which w_j is a keyword. (That is, w_j is a keyword in q_i exactly when $i \in \{i_{j,1}, \ldots, i_{j,\ell(j)}\}$.)
4. The server computes the Identity-Based Encryption c_j of the string

   $$(\mathbf{flag}|i_{j,1}|K_{i_{j,1}}|\cdots|i_{j,\ell(j)}|K_{i_{j,\ell(j)}})$$

   using w as the public key and P as the public parameters.
5. The server writes $E_{K_1}(m_1), \ldots, E_{K_t}(m_t), c_1, \ldots, c_u$ as the block and index to the audit log.

As the length of the IBE-encrypted strings grow we may use hybrid encryption for efficiency: for a long string M we compute the IBE encryption a one-time symmetric key K_0, then perform block encryption of M using the symmetric key K_0. (This was not necessary in the non-indexed case, as the strings encrypted were very short.)

Indexing introduces a significant performance advantage for searching/decryption when keywords are repeated among several audit log entries within a block. When a keyword w is present in k entries in a log block, only one pairing operation and one modular exponentiation are required to find and decrypt the k audit log entries.

Using indexing also results in a big performance win for audit log generation. For a keyword w appearing k times in a block, again only one pairing operation and one modular exponentiation are required to generate the index entry relevant to w.

We note that this method may open up a slight vulnerability: an attacker may obtain partial information about the frequency of keywords present in a single block by observing the lengths of the IBE encrypted strings within the index. This can be thwarted by adjusting the block size to be small enough to limit the amount of statistical knowledge obtained (which, in the limit of $t = 1$, reduces to the security of the non-indexed solution.)

RANDOMNESS REUSE. Lastly, we consider an optimization for the decryption process. We perform an independent IBE encryption to creating the c_i corresponding to the

optimization method	encryption		search/decryption
	pairings	exponentiations	pairings
none	$t \cdot v$	$t \cdot v$	$t \cdot v$
pairing reuse (PR)	u	$t \cdot v$	$t \cdot v$
indexing	u	u	u
randomness reuse (RR)	$t \cdot v$	$t \cdot v$	t
PR + RR	u	$t \cdot v$	t
all three	u	u	1

Table 1. Number of compute-intensive operations needed to process a block of t log entries, including in total u distinct keywords, with an average of v keywords per log entry.

keywords w_i for given log entry. However, it is possible to reuse an intermediate result of the IBE encryption process: we may save the value r chosen in step (3) of the encryption that produces c_1 to use in calculation of $c_2, \ldots, c_n$. As long as the w_i are distinct keywords, this reuse of the randomness produces results indistinguishable from the original method. This speeds up decryption, as only one pairing is needed for each distinct r chosen. This implies that instead of n pairings required to test if any of the c_i match a given keyword, only 1 pairing is required.

OPTIMIZATION SUMMARY Table 1 summarizes the number of compute-intensive operations to process (encrypt or search/decrypt) a block of audit log entries. Table 2 summarizes the storage requirements of a block of audit log entries.

6. Implementation

We implemented a database audit log system that creates asymmetrically encrypted and searchable entries. The logger is implemented as a MySQL proxy server. The user signs onto the proxy and makes SQL queries. The proxy server, upon receiving a query, logs the query in addition to passing it to the MySQL database server.

The proxy was developed on a Linux platform and is multi-threaded so that multiple users can be served simultaneously and that the logging component runs in parallel with the rest of the system. The audit log server attaches the date and time to the audit log entry. The log entries are written to another MySQL database server that is dedicated to storing audit log entries.

We used the Stanford Identity-Based Encryption library [12] for the basic IBE operations[5] and the Cryptlib library [5] as the implementation of the the symmetric encryption of the query itself. We parameterize IBE with values $p = 1024$ and $q = 160$. We use a 128-bit AES key for the symmetric encryption.

The server software has a cache that is used to reuse pairings as described in Section 5.3. The cache is implemented as a simple hash table which associates the pairing result g_w with the keyword w. Every time a keyword w that has not been seen before is used, the newly computed pairing g_w is stored in the hash table. Another optimization we implemented is the reuse of randomness described in Section 5.3.

We also implemented the hash chain method of checkpointing described in Section 4. The audit log server computes the updated value of the hash chain for every audit log entry it constructs. The current hash value can be read at any point in time. A party which reads this value can use it later to check the integrity of the audit log for all entries written before the hash checkpoint.

Finally, we implemented the tool the investigator uses to search the audit log when given a capability. The tool retrieves all records from the audit log and searches them one record at a time. As mentioned above, our implementation uses the same randomness for each encryption within an entry. Therefore, searching an entry only requires one IBE pairing. In future work, we plan to implement the indexing algorithm described in Section 5.3.

Our performance measurements were taken on a Pentium IV processor machine running RedHat Linux 9.0 with 2GB of memory. The speed of the processor is 2.8GHz.

We measured the added cost of encryption for each searchable keyword that is part of the query. If a keyword w does not have a corresponding cache entry g_w, then the server must hash w into the group $\mathbb{G}_1$, execute a pairing to compute g_w, and also compute a modular exponentiation. The cost of these operations totals 180ms. However, if there is a cache entry for g_w, the server only needs to execute an exponentiation and the cost is 5ms.

Clearly, the use of the cache is important for efficient operation of the system. We expect that in most applications most extracted keywords will have a corresponding public key in the cache (provided the system has been running for a sufficient amount of time). We also do not anticipate memory limitations to effect most caches. A 100MB cache can hold approximately $800,000$ public keys. In compari-

[5] We note that an implementation of IBE that is approximately twice as fast has recently become available as part of the miracl package [10].

optimization method	storage requirement (in bits)
none	$t \cdot (M + v \cdot \log_2 p + v \cdot n_{H_2})$
indexing	$t \cdot M + u \cdot (\log_2 p + n_{H_2}) + t \cdot v \cdot \log_2 t$
randomness reuse (RR)	$t \cdot (M + \log_2 p + v \cdot n_{H_2})$
indexing + RR	$t \cdot M + \log_2 p + u \cdot n_{H_2} + t \cdot v \cdot \log_2 t$

Table 2. Storage requirements of a block of t log entries, including in total u distinct keywords, with an average of v keywords per log entry. M is the average bit length of a log entry, p is the prime used for IBE operations, and n_{H_2} is the output bit length of the hash function H_2 used for IBE operations. Pairing reuse has no effect on storage requirements.

son the number of entries in the second edition of the Oxford English Dictionary is approximately $300,000$.

The tool which searches the encrypted audit log must compute a pairing per entry. This operation takes 81ms.

7. Conclusion

Designing a secure audit log is not a trivial task. Apart from guaranteeing properties such as tamper resistance and verifiability, the contents of the audit log may itself be considered sensitive, and need to be protected from unauthorized access.

A natural approach to such protection is to encrypt the audit log, which needs to be done in such a way that the log still remains effectively searchable. We presented a scheme in which we use identity-based encryption to protect symmetric keys that are used to encrypt audit log entries. Privileged *audit escrow agents* can create search capabilities that allow their bearer to search the audit log for records matching certain keywords.

We implemented our scheme as a secure audit log for MySQL database queries. It turns out that the identity-based encryption scheme we use introduces considerable overhead (although small enough to be negligible in an interactive system), but it buys us security and convenience over symmetric key based schemes.

Our current implementation relies on checkpointing to secure the integrity and verifiability of the audit log. While the focus of our work so far has been to investigate the searchability of the audit log, we plan to implement more advanced integrity protection mechanisms to improve the overall security of the system.

8. Acknowledgments

This work was sponsored by DARPA grant F30602-03-C-0037.

References

[1] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. *Lecture Notes in Computer Science*, 2248, 2001.

[2] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Searchable public key encryption. Submitted for publication. See `http://eprint.iacr.org/2003/195/`.

[3] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Proc. CRYPTO 01*, pages 213–229. Springer-Verlag, 2001. LNCS 2139.

[4] E.-J. Goh. Building secure indexes for searching efficiently on encrypted compressed data. Submitted for publication. See `http://eprint.iacr.org/2003/216/`.

[5] P. Gutmann. cryptlib. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[6] S. Haber and W. Stornetta. How to time-stamp a digital document. In A. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 437–455. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.

[7] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th USENIX Security Symposium*, pages 53–62. USENIX Press, 1998.

[8] B. Schneier and J. Kelsey. Minimizing bandwidth for remote access to cryptographically protected audit logs. In *Web Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection*. USENIX Press, 1999.

[9] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[10] Shamus Software Ltd. MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library. `http://indigo.ie/~mscott/`.

[11] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[12] Stanford Applied Cryptography Group. IBE secure e-mail. `http://crypto.stanford.edu/ibe`.