SELL v1.0: Searchable Encrypted Logging Library

Amir Jalali Neil Davenport

LinkedIn Corporation
Sunnyvale, CA
{ajalali, ndavenport}@linkedin.com

Abstract. We present a practical solution to design a secure logging system that provides confidentiality, integrity, completeness, and non-repudiation. To the best of our knowledge, our solution is the first practical implementation of a logging system that brings all the above security aspects together. Our proposed library makes use of a Dynamic Searchable Symmetric Encryption (DSSE) scheme to provide keyword search operations through encrypted logs without decryption. This helps us to keep each log confidential, preventing unauthorized users from decrypting the encrypted logs. Moreover, we deploy a set of new features such as log sequence generation and digital signatures on top of the DSSE scheme, which makes our library a complete proof of concept solution for a secure logging system, providing all the necessary security assurances. We also analyze the library's performance on a real setting, bootstrapping with 10,000 lines of logs. Based on our observation, the entire search operation for a keyword takes about 10 milliseconds. Although SELL v1.0 is developed purely in Python without any low level optimization, the benchmarks show promising timing results for all the operations.

Keywords: Secure logging, dynamic searchable symmetric encryption, privacy-preserving, information security

1 Introduction

Preserving privacy for a user's data has become a serious concern for many companies and data storage services recently. New regulations enforce several restrictions on the user's data management such as the purpose of data processing, the amount of data that can be collected, the storage time, and the entities who can access data. Though all these rules protect data, giving data owners more control over their personal information, the new regulations also present challenges for design and implementation of privacy-preserving applications.

Authorized access to data compliance is not a new concept in the security field. Over time, different cryptography protocols have been designed and developed to ensure that sensitive data can only be accessed by authorized users through encryption/decryption and can only be decrypted by those who possess the key. In the data mining era, however, every piece of data can be interpreted

as a sensitive asset that should not be accessed or used without the owner's permission. This leads to a complicated situation where developers using any data must comply with different regulations to address every security concern. Accordingly, during the last decade, privacy-preserving solutions such as *homomorphic encryption*, the state-of-the-art cryptography primitive that enables the arithmetic operations on encrypted data, have received more attention. The performance and usability of such schemes however are still not efficient enough to be widely adopted.

A work-around to this problem is to restrict the required operations on the encrypted data so a more efficient and practical solution can be developed. Although this strategy cannot be applied to many applications, some applications may benefit significantly from losing the generality. In particular, logging systems are one of the applications that only require a certain set of operations, such as keyword search on the stored logs, that can be performed fairly efficiently even on encrypted data.

1.1 Problem Statement

One of the objectives of a privacy-preserving environment is to remove access to raw data or to restrict it to authorized users. This necessitates a framework that is able to perform a set of required operations on the data without revealing any information about the identity or values of the data. Adopting this methodology in a logging system, we require a framework that bolsters the confidentiality, integrity, and completeness of logs (either system or user logs) at a specific level of security, while the required operations, such as keyword search, are still possible. In this work, we concentrate on the design and development of a software library that provides a secure solution for generating, storing, and searching through the *sensitive logs*. Since the nature of logging systems is streaming, we adopt a dynamic structure that enables the log addition and deletion to and from an encrypted database, respectively. We adopt the dynamic method by Kamara et al. [6], which provides a fully dynamic searchable encryption on files; however, we customize and add new features to this method to fit it into our specific purpose, providing the security requirements. Our proposed solution is fully customizable. All the cryptography primitives can be replaced by other schemes depending on the target level of security and communication overhead. We refer the readers to section 4 for further details.

1.2 Roadmap

In section 2, we give a brief overview of DSSE scheme and its related methods. In section 3, we present our proposed solution is presented to address the security concerns in the logging system. We describe the implementation details as well as API and workflow of the SELL v1.0 in section 4. In section 5, we present the performance observation of the library with detailed timing for each operation and discuss different strategies to improve the performance. We conclude this work and explore the possible future work in section 6.

2 Preliminaries

Searchable Symmetric Encryption (SSE) is the most relevant primitive for developing a system that can search through encrypted documents. Although other schemes that are based on public-key cryptography provide the same functionality and possibly better key distribution models, we developed our library based on symmetric encryption due to performance reasons that are a necessary factor in logging systems. In this section, we briefly describe the DSSE scheme that we adopt inside our library. We refer the readers to [6] for further details on the algorithms and the security of the scheme.

2.1 Searchable Symmetric Encryption

SSE is a scheme that allows the ability to perform keyword searches on encrypted data. In the literature, there are two main approaches available for performing such operations. The first approach, proposed by [5], computes an encrypted index for each document,[1] which contains the information about the occurrence of keywords in the document. In a search query, the encrypted index for each document is tested for the given keyword. Obviously, this method is not efficient in any sense compared to an inverted index, which led to the second method of SSE proposed by Curtmola et al. [3]. The SSE schemes based on inverted indexes contain a single encrypted index that maps all the available keywords inside all the documents to the document identifiers that contain those keywords. This method is the most efficient way of searching and it is the fundamental approach used by different search engines. Several works [10,2,12,9] studied the usage of SSE from different aspects. While most of these approaches are secure and practical, they are not suitable in the context of dynamic applications such as logging systems due to the static construction of the encrypted index.

2.2 Dynamic Searchable Symmetric Encryption

The only main difference between SSE and DSSE is the ability to dynamically add and remove a document from the encrypted index. Moreover, the update operation should not leak any information about either the keywords or the documents. Recently, different approaches on DSSE have been proposed with different sets of functionality and performance metrics; however, logging systems are streaming applications by nature. Hence, we prefer to apply an efficient and practical solution that provides the desired level of security in an environment with a semi-honest server model.

The proposed DSSE method by Kamara [6], provides a solid security proof against adaptive indistinguishable Chosen Keyword Attacks (IND-CKA2). This implies that, given access to an encrypted index, the adversary cannot learn

[1] Document in SSE is an entity with a label that contains a set of keywords; the keyword search operation returns the document label.

any information about the index entries and the encrypted keywords. This security assumption is very conservative in the logging system with semi-honest adversary model where the encrypted index holder (i.e., collector) merely tries to gather information out of the protocol, but does not deviate from the protocol specification. We discuss the security of our library further in section 3.5.

We briefly present the main operations of the DSSE scheme [6] that we include inside our library in the following. Later in section 4, we describe our implementation details for each of these operations.

1. **Key Generation:** Generates the symmetric key under which the logs and the keyword index are encrypted. The DSSE scheme that is used in this library has four separate secret-keys: $\texttt{sk}_1, \texttt{sk}_2, \texttt{sk}_3, \texttt{sk}_4$. In addition, we need an extra secret-key `sk_sign` for the generator's signature. All of the keys are randomly generated according to the given security parameter k.

$$\texttt{sk}_1, \texttt{sk}_2, \texttt{sk}_3, \texttt{sk}_4, \texttt{sk\_sign} \in \{1, 0\}^k \tag{1}$$

2. **Encryption/Decryption:** The symmetric encryption/decryption algorithm that is used to encrypt and decrypt logs, respectively. In our software, $\texttt{sk}_4$ is used for this purpose.

$$\texttt{Enc}(\texttt{sk}_4, \texttt{log}) \rightarrow \texttt{log.enc}, \quad \texttt{Dec}(\texttt{sk}_4, \texttt{log.enc}) \rightarrow \texttt{log} \tag{2}$$

3. **Search Token Generation:** This operation generates a search token from a given keyword using a secret-key. The generated search token must not reveal any information about the input keyword and it should be secure against adaptive Chosen Ciphertext Attack (CCA2). Given a keyword w, the associated search token is generated using $\texttt{sk}_1, \texttt{sk}_2$, and $\texttt{sk}_3$ as follows:

$$\texttt{srch\_tkn} = (\texttt{HMAC}(\texttt{sk}_1, w), \texttt{HMAC}(\texttt{sk}_2, w), \texttt{HMAC}(\texttt{sk}_3, w)) \tag{3}$$

4. **Add Token Generation:** Generates an add token from a given log and a secret-key. The generated add token is sent to the collector and all the keywords inside the log are added to the encrypted index without revealing any information about the keywords. The generated add token is produced by $\texttt{sk}_1, \texttt{sk}_2$, and $\texttt{sk}_3$.

$$\texttt{add\_tkn} = (\texttt{HMAC}(\texttt{sk}_1, \texttt{log}_{\texttt{id}}), \texttt{HMAC}(\texttt{sk}_2, \texttt{log}_{\texttt{id}}), \lambda_i), \tag{4}$$

   where $\texttt{log}_{\texttt{id}}$ is the log identifier and λ_i is a set of encrypted values associated with each keyword inside the log ($i = \#w$). Each λ_i is generated using exclusive-or and the concatenation of keywords and log identifier, which are hashed using $\texttt{sk}_1, \texttt{sk}_2$, and $\texttt{sk}_3$. We refer the readers to [6, Figure 3] for further details.

5. **Delete Token Generation:** Similar to add token, this procedure generates a delete token using a given log and a secret-key. The generated token is sent to the collector and deletes all the keywords that are included inside

the deleted log without leaking any information to the collector. This value is computed as:

$$\texttt{del\_tkn} = (\texttt{HMAC}(\texttt{sk}_1, \texttt{log}_{\texttt{id}}), \texttt{HMAC}(\texttt{sk}_2, \texttt{log}_{\texttt{id}}), \texttt{HMAC}(\texttt{sk}_3, \texttt{log}_{\texttt{id}}), \texttt{H}(\texttt{log}_{\texttt{id}})) \tag{5}$$

6. **Search:** The search operation is performed by the collector given a generated search token. During this procedure, the collector searches for the given keyword (encrypted) and returns all the log identifiers that include the keyword. Upon receiving the generated search token (3), the collector unpacks three `HMAC` digests. The first hash value (generated using $\texttt{sk}_1$) is associated to the inverted index *keys*. If this value exists, the collector yields the corresponding log IDs, otherwise the keyword is not in the index. In order to bolster security against search pattern attacks, the log IDs are randomly allocated inside the inverted index and each node contains the address of next list node. See [6, Figure 3] for further details.

7. **Add:** Given an add token, the collector adds all the keywords inside the add token to the current encrypted index without learning any information. When the collector receives an add token (4), it unpacks the hash digests and all the λ_i values. Subsequently, the collector discovers the address of an addition node for each λ_i inside a search table and updates the index.

8. **Delete:** Given a delete token (5), the collector removes all of the included keywords out of the encrypted index without learning any information. Similar to the addition operation, the collector decodes the address of deleted keywords by unpacking the hash digests and stepping through the delete table.

Considering the above operations, we are able to design a dynamic secure logging system that preserves the confidentiality of log data while enabling the authorized entity to search for a keyword and retrieve the target logs. However, preserving the confidentiality of the data is not enough for a full verifiable and secure logging system. Therefore, we propose additional features on top of the DSSE scheme to bolster the integrity, completeness, and authenticity of the logs. We explain our proposed solution in the next section.

3 Proposed Solution

In this section, we describe our methodology for providing all the required security properties of a verifiable and secure logging system. We carefully engineer the available solutions and employ the most compatible ones in a single software library to provide security assurance of log data inside a system.

3.1 Confidentiality and Privacy

The main objective of this project is to ensure the confidentiality of each log. In most logging infrastructures, user logs are stored as raw data inside a database.

Any entities who have access to this database can read and interpret all the log data despite perhaps not being authorized. To solve this problem, all the logs are encrypted using a symmetric encryption scheme and only parties who possess the secret-key can decrypt and read the data. Using a secure symmetric encryption scheme guarantees the confidentiality and privacy of log data as long as the shared secret-key is distributed securely. We discuss the encryption scheme that we use inside the library in section 4.

3.2 Integrity

We define the log integrity assurance as the ability to verify that the generated log is not manipulated or tampered with during transmission. This property is an essential concern in environments where the communication links are not secure and vulnerable to man-in-the-middle attacks (MITM). In this scenario, an attacker secretly alters the communication between the log generator and log collector. Using an *authenticated encryption* method which offers a Message Authentication Code (MAC) in addition to encrypting messages provides both authenticity and integrity of the data. In our solution, we adopt the `AES_GCM` authenticated encryption method for encrypting the logs and generating a MAC tag, that ensures both integrity and authenticity of the logs.

3.3 Completeness Check

The RFC5848 [8] describes a mechanism to sequence and detect missing messages in the transmitted syslog using a signature block. Inspired by that, we design and develop a log sequencing mechanism that encapsulates the information corresponding to each log, such as log ID and application name, in addition to its parent log ID. The generated log sequence is verified at the collector upon receiving each log and any mismatch in the sequence hash illustrates incompleteness. The incomplete logs are recorded inside a list at the collector and can be accessed later to check the log completeness. To mitigate the threat of MITM attacks on log sequence values, the log generator also digitally signs the information that comprises the log sequence. The generated signature is verified at the collector prior to a sequence check to ensure the authenticity and integrity of the log sequence.

Note that, since the collector does not possess the symmetric key, it is not possible to integrate the log sequence inside the log stream and encrypt the entire data using authenticated encryption.

The above method checks for the completeness of logs and detects the missing intervals for in-order delivery[2]; however, there are situations where logs are transmitted over UDP, which leads to out-of-order distribution. To overcome this shortcoming, RFC5848 [8] defines a Global Block Counter (GBC) that determines the number of signature blocks sent prior to the current one. This value

[2] Such applications use TCP or TLS connections to ensure the sequence order of logs at the transport layer.

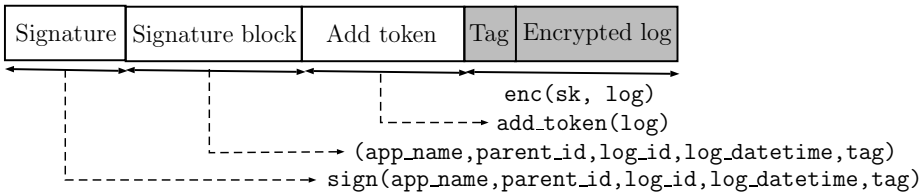

Fig. 1: Generated add payload by log generator

increases from 0 to 9,999,999,999 incrementally and resets when it reaches the maximum value. In this situation, RFC5848 defines another variable, session ID, which is increased incrementally whenever the GBD reaches the maximum value. The session ID informs the collector that the sequence number was reset.

SELL v1.0 does not support such functionality and it only supports the completeness check over the frameworks where in-order delivery is handled by the transport layer. Supporting the above mechanism is not complicated and only requires a small change in the payload structure; however, we believe sequence check for out-of-order delivery adds a considerable load on the collector, which leads to overall performance degradation.

3.4 Non-Repudiation

Non-repudiation is a high assurance of log authenticity that is very useful during the audit procedure and proof of events. As mentioned above, the log generator signs each log sequence with a signing key and the signature is verified on the collector using the generator's verification key. Moreover, each log is encrypted using authenticated encryption and the authentication tag is included in the Signature block. The log authenticity can also be checked by the reviewer during the decryption process by querying the collector for the verified signature block of the log id which includes the authentication tag. The distribution of the generator's public-key depends on the available applications and infrastructure; in the current version of the library, the generator's public-key is passed to the collector's constructor as an argument. Alternatively, the collector can retrieve the valid public-key from a Trusted Certificate Authority (TCA).

Since the digital signature provides the authenticity of the origin, our proposed solution also provides non-repudiation assurance since the collector can assert the digital origin of a log.

Encrypted log, add token, signature block, and signature are encapsulated in a single payload and sent to the collector as a new added log. Fig. 1 illustrates such a payload. Note that the encrypted log and tag can be excluded from the payload and sent to a third-party storage; the collector does not require this part for the search operation. Accordingly, the log reviewer can download the target encrypted logs (their IDs are retrieved from the collector) from the third-party database and decrypt them locally with their secret-key.

The `log_datetime` value that is included in the payload is used for time-specified search operations. We explain this feature in detail later in section 4.5.

3.5 Security

As well documented in the literature, there are several common attack models to consider when evaluating the security of an encryption scheme[1]. The indistinguishability game is defined as follows:

1. A challenger generates a key pair, keeping the secret-key private and providing the public-key to the adversary $\mathcal{A}$.
2. The $\mathcal{A}$ can perform any number of decryption and encryption operations using the provided public-key (for encryption) and the decryption oracle.
3. The $\mathcal{A}$ then deliberately chooses two messages m_0, m_1 which they believe will give them the best advantage in the game. Then sends both of these messages to the Challenger.
4. The Challenger then randomly chooses a bit from the set $b \leftarrow \{0, 1\}$.
5. Based on the value of this bit, the Challenger then encrypts either message m_0 or m_1 producing $C_b \leftarrow Enc(sk, m_b)$.
6. Before receiving C_b from the challenger, the $\mathcal{A}$ can perform any additional operations that they so choose.
7. The $\mathcal{A}$ then receives C_b from the Challenger.
   - In the non-adaptive (IND-CCA1) scenario, the $\mathcal{A}$ cannot make any further calls to the decryption oracle.
   - In the adaptive (IND-CCA2) scenario, the $\mathcal{A}$ *can* perform additional decryption operations, but may **not** submit the C_b to the decryption oracle, as this would provide the solution to the game thus invalidating it.
8. Now the $\mathcal{A}$ makes a guess as to the value of b' out of the possible set which was $\{0, 1\}$.

We say that the scheme is IND-CCA1/CCA2 secure if

$$\text{Prob}[b = b'] \leq \frac{1}{2} + \epsilon(k), \tag{6}$$

where $\epsilon(k)$ is a negligible function in the security parameter k.

In layman's terms, this means that the $\mathcal{A}$ has almost no advantage over random guessing. That is $\frac{1}{2}$.

In addition to the underlying symmetric encryption, digital signature, and hash functions, DSSE schemes should be secure against the related attack models. According to [6,3], the highest level of security for an SSE scheme is defined by IND-CKA2. We describe this concept in the context of logging systems in the following [3]:

1. An adversary $\mathcal{A}$ generates a collection of logs, $\texttt{logs} = (\texttt{log}_1, \cdots \texttt{log}_n)$.
2. The challenger generates a secret-key $\mathcal{K}$ randomly based on a security parameter k and the corresponding encrypted indexes from logs: $\texttt{Index} = (I_1, \cdots, I_n)$ where $I_i \leftarrow \texttt{Index}_{\mathcal{K}}(\texttt{log}_i)$. Note that, here $\texttt{Index} = (I_1, \cdots, I_n)$ is an encrypted set based on the secret key $\mathcal{K}$ and it is public. The goal here is to evaluate the security of a scheme by assessing the adversary's ability to find any relation between an encrypted index and the corresponding plain log.
3. Given a full access to $(I_1, \cdots, I_n)$ and *token-oracle*[3], $\mathcal{A}$ chooses two logs $\texttt{log}_0^* \in \texttt{logs}, \texttt{log}_1^*$ after an arbitrary polynomial order of computations. Moreover, $\mathcal{A}$ is not allowed to generate any token from the keywords in $\texttt{log}_0^*, \texttt{log}_1^*$. $\mathcal{A}$ sends these logs to the challenger.
4. The challenger chooses a random bit $b \leftarrow \{0, 1\}$ and generates $I_b \leftarrow \texttt{Index}_{\mathcal{K}}(\texttt{log}_b^*)$ and sends I_b back to $\mathcal{A}$.
5. Given I_b and access to the token-oracle even after receiving I_b (adaptive), $\mathcal{A}$ outputs $b' \in \{0, 1\}$ without generating any token from the keywords in $\texttt{log}_0^*, \texttt{log}_1^*$.

The scheme is IND-CKA2 (Chosen Keyword Adaptive Attack) secure if for every polynomial time computations performed by adversary $\mathcal{A}$:

$$\text{Prob}[b = b'] \leq \frac{1}{2} + \epsilon(k), \tag{7}$$

in other words, the adversary has a negligible advantage over random guessing. The underlying DSSE scheme is secure against IND-CKA2. The detailed security proof of the algorithm is described in [6, Section 5]. Since the scheme is IND-CKA2 secure, we expect that a semi-honest collector cannot obtain any information about the keywords and log data except for the number of logs and the number of keywords inside the encrypted index.

In addition to IND-CKA2, collectors should not be able to learn the search pattern for a specific keyword. Although, the search pattern attacks cannot compromise the confidentiality, after a certain number of search operations for a keyword, the collector can learn the logs that include the keyword. The DSSE scheme in [6] addresses this problem by constructing a fully random assignment of elements to the inverted index. While this approach degrades the overall performance because of the lack of locality, it prevents the collector from learning from the access pattern. In the next section, we describe the methodology that we use to implement different features inside our library. SELL v1.0 is publicly available[4].

4 Implementation Summary

In this section, we describe our implementation methodology. Our proposed library is designed as a proof of concept and it can be customized easily for dif-

[3] $\mathcal{A}$ can generate any add/search/delete token from the chosen input.
[4] Our library will be publicly available soon.

ferent applications and environments. In particular, the `keyword extraction` method can be simply replaced by a custom version. In the current version, all the words inside a log separated by a space are extracted and attached to the encrypted index. While this covers all the possible existing whole keywords, it increases the size of the encrypted index, which consequently leads to performance loss. Alternatively, the keyword extraction can be performed only on the specific parts of a log data such as user, date, action, protocol, and so on. This will significantly improve the efficiency of the library since the encrypted index size is optimal.

4.1 Implementation Parameters

This secure logging library consists of different cryptographic modules such as hash functions, HMAC, symmetric encryption, etc. Each one of these modules directly affects the overall performance and security of the library. We chose a set of default values for implementation parameters that yield reasonable performance and security properties for the scheme. We briefly explain these parameters in the following:

Symmetric Encryption. As discussed earlier, we use an instance of `AES_GCM` authenticated encryption inside the library to ensure integrity. In particular, the default parameter generates a 256-bit secret-key and 128-bit `iv` for the encryption procedure, providing a 256-bit security level. These parameters are in compliance with NIST recommendations [4] for the GCM mode. Alternatively, users can adopt other authenticated encryption modes such as OCB, CCM, or CWC.

Hash Functions. The underlying DSSE scheme makes use of three HMACs with different keys. We use an instance of `HMAC_SHA256` for the default implementation of these functions, which is recommended by RFC4868 [7]. Moreover, the log ID and log sequence are also hashed using `SHA256`, providing 128-bit and 256-bit security against collision and preimage attacks. Furthermore, we use `BLAKE2b` hash for generating variable length hash digests.

Digital Signature. All the generated sequence hashes are signed with the log generator's signing key and verified by the collector. We adopt `ECDSA-NIST384p`, which in combination with `AES_GCM-256` provides the `SUITE-B-GCM-256` security level [11]. Since asymmetric cryptography operations such as signing and signature verification are much slower than symmetric cryptography, increasing the security level of the digital signatures may affect the performance of the library considerably. If the security requirement allows for lower levels, we highly recommend users to deploy smaller parameter sets such as `NIST256p` or `NIST192p`, which significantly improves the performance of the library.

4.2 Classes and APIs

In a logging system, different log generators produce logs based on actions and send them to a central repository. This gives us the ability to search through the logs and to detect and describe an incident. In this architecture the log generator, collector, and log reviewer are separate entities with different functionality. Therefore, in the OOP terminology, each one of them can be encapsulated into a unique class. We designed our library based on this architecture, which includes three main classes: `Generator`, `Collector`, and `Reviewer`[5]. We explain each of these classes and their main methods in the following:

Generator Class. This class is designed to generate the encrypted payload from the logging application/system. The constructor generates all the required secret-keys according to the given security level.

This class contains a `bootstrap` method that accepts a list of log files as the input. The `bootstrap` method generates the initial encrypted index from the given log files. The `collector` constructor later gets this bootstrap index as the input to construct the main tables for the `Generator` object. In order to add a new log to the collector encrypted index, `Generator` can call either `file_add_token` or `line_add_token`. These methods have the same functionality and generate the payload, which is illustrated in Fig. 1 according to the log structure. See section 4.4 for further details.

Collector Class. This class contains the encrypted index that is generated by `Generator` objects. The constructor of the class gets the initial encrypted index from the bootstrapping phase. The `Collector` class offers `search`, `add`, and `delete` operations upon receiving a `search_token`, `add_token`, and `delete_token`, respectively. In particular, an add token is generated by `Generator`, while search and delete tokens are produced by `Reviewer`. `Collector` is also responsible for justifying the completeness and authenticity of the incoming logs using hash and signature verification methods. It stores a list of incomplete sequences, as well as fake signatures, which can be easily accessed by `Reviewer` during the audit procedure.

Reviewer Class. This class can search for keywords and also delete logs from the encrypted index. The constructor takes the generated secret-key from the `Generator` class.

Note that the key-exchange procedure between `Generator` and `Reviewer` classes is out of scope of this library. This procedure can vary in different infrastructures based on the availability of secure protocols. For instance, a simple static Diffie-Hellman key exchange can provide a shared secret-key between `Generator` and `Reviewer` objects. It is also possible that these two objects exist on a single machine where no key-exchange method is required.

In addition to `search_token` and `del_token` methods, we design a special feature for this class, `count_token`, which can query the number of occurrences

[5] The class names are in compliance with RFC5848 notations.

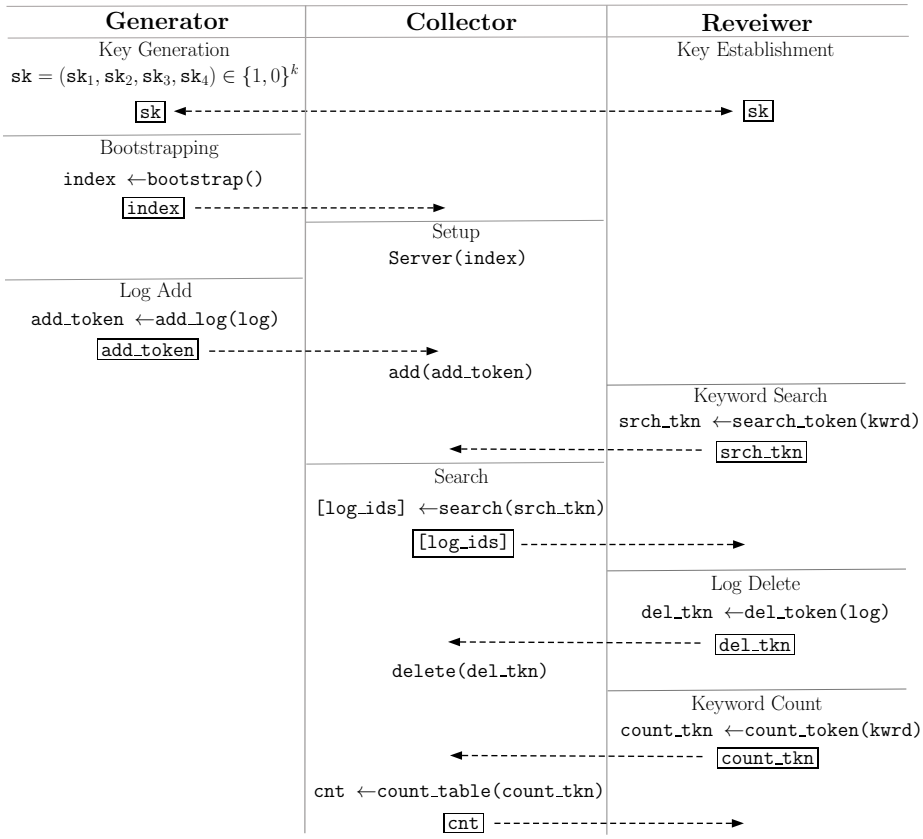

Fig. 2: SELL v1.0 workflow

of a keyword inside the entire log repository. Based on the required operations in a logging system, we find it very useful to have this ability without decrypting the logs. For instance, the occurrence count can be used inside a SIEM appliance or a dashboard to instantly illustrate the number of occurrences of a special keyword.

4.3 Workflow

In this section, we describe the workflow of our presented library in a nutshell. We consider a single instance of log generator, collector, and log reviewer in this model; however, one can easily extend this model to multiple generators and reviewers as long as a secure key distribution mechanism is applied.

Alternatively, the collector and secure index can be extended to multiple instances and corresponding operations can be performed on each instance separately and accumulated at the end in a familiar map reduce pattern. Fig. 2 presents the workflow of our library. Different operations are separated by labels

and the corresponding payloads are defined in the box. As mentioned before, the `add_token` payload includes more information than the addition token to enable the completeness check. In terms of overall performance, operations such as bootstrapping and search require more instructions compared to other operations. Note that retrieving the number of occurrences of a keyword is a simple search through a hash map table. Therefore, this feature can be easily deployed in real-time event monitoring applications.

4.4 Different Log Structures Support

The described DSSE method in [6] is designed for keyword search inside a file. While system logs can be stored as a batch inside a log file, in some situations, single line streaming logs need to be encrypted and sent to the collector. Therefore, we implement this functionality inside the library using a set of different APIs. Based on the log structure, either file or line encryption methods can be called by the log generator and the corresponding keywords are extracted from a file or a single line, respectively. Note that, in the log file encryption scenario, all the log lines inside the file are identified by a single log ID, which is the log `filename`.

4.5 Efficient Time-Specified Search

Since SELL v1.0 is designed to provide a secure logging framework for various applications, we design and develop the necessary features to allow the reviewer to perform investigation operations as efficiently as possible.

Searching for a keyword inside a specific time period is a common operation in an investigation. This requires each log to include timing information that can be later searched for. We propose a new method that integrates timing information for a log inside the encrypted index. Our proposed solution adds extra memory complexity on the collector, but it provides a notable performance improvement for the keyword search operation.

In contrast to an unencrypted logging database where the timing information is assigned to each log and the search operation is performed on each single log entry, in SELL v1.0 the search index is an inverted index with keywords as its key labels. Moreover, log IDs are allocated inside a dynamic linked list where the address of the next node is encrypted inside the previous node. This kind of data structure inherently requires sequential steps through the entire list to retrieve all the log IDs. Therefore, we include a time-stamp with each log ID and store them as a 2-tuple in the collector's ID database. Note that the time-stamp is generated by a generator using `datetime` package[6]. During the search procedure, each yielded log's time-stamp is checked to ensure it is in the specified search time-range. Since the log IDs are stored in *chronological order*, the last received log containing the searched keyword is retrieved first from the encrypted index. This provides us with an optimal search strategy: each recovered log ID

[6] Alternatively, a user can customize the time value and extract it from the log data.

Table 1: Benchmark results of SELL v1.0 for different operations. Benchmarks are obtained on a 3.0 GHz Intel Core i7 running macOS v.10.13.3. Numbers represent the average of 1,000 iterations and are presented in milliseconds. The loaded log file contains 10,000 Apache log lines. The log line benchmark is performed on 10 lines of logs iteratively.

Operation	**Log File (ms)**	**Log Line (ms)**
Bootstrapping	40,701	61
Add Token	4,442	6
Search Token	2	
Delete Token	3	
Search	8	
Add	6	
Delete	11	

is compared with the beginning of the search time-slot and if it is older than this value, the search operation is terminated. In order to be immune to any manipulation of log time-stamps, as illustrated in Fig. 1, this value is included inside the signature block and signed by generator.

In SELL v1.0, the `search_token` method can include `start_time` and `end_time` for a keyword. These arguments are optional and can be replaced with default values if the user does not specify time intervals for the search.

5 Performance and Discussion

5.1 Performance Evaluation

In this section, we report the performance of the secure logging library on a practical setting to evaluate the efficiency and feasibility of our proposed solution. The reported benchmarks are generated based on default implementation parameters, which are discussed in section 4.1, providing `SUITE-B-GCM-256` level of security. Lower levels of security will result in much better performance metrics due to the smaller digests and signatures.

We benchmark the library on a MacBook Pro equipped with 3 GHz Intel Core i7 running macOS v.10.13.3. The library is developed in Python and benchmark tests are run with Python 3.6.1[7]. Table 1 presents the performance timing of the SELL v1.0 for different operations. The benchmark results are obtained using `pytest-benchmark` and are represented in milliseconds. For each operation, we benchmark the library over 10 rounds with 100 iterations each and present the average time. Note that the performance timings are directly related to processor's frequency. Faster processors can improve these numbers considerably.

[7] SELL v1.0 also supports Python 2.7.

We instantiate the collector with an encrypted index loaded with 10,000 lines of logs at the bootstrapping phase. The sample logs are generated by `Fake Apache Log Generator`[8]. For the `file_add_token` operation, a log file containing 10,000 log lines is added to the collector. Based on the timing results, adding the log file is more than 10 times faster than bootstrapping a file with the same size. Search, add, and delete operations are also performed on the index of 10,000 logs.

The current version of SELL is implemented purely in Python without taking any advantage of optimization from Cython or C++/C implementation. We plan to implement these optimizations in the future versions of the library and expect to see significant performance improvements.

5.2 Discussion on Performance Improvement

The reported performance in the previous section is associated with the default parameter set that we use in SELL v1.0. Moreover, the keyword extraction method simply extracts all the unique words inside a log. Although, this seems to be a reasonable assumption for many applications, we highly recommend extracting `metadata` from the log data instead of all possible words. This results in an optimal encrypted index in size, which leads to more efficient search, add, and delete operations on the collector.

Furthermore, the keyword extraction method can be implemented using efficient searching techniques, such as regular expression matching with an n-grams index. In particular, Google Code search engine benefits notably in terms of performance from 3-grams indexing[9]. However, there is a trade-off between the search-time improvement and the redundant memory complexity of the generated n-grams index. As mentioned before, the keyword extraction method is totally customizable and developers can implement it based on their application requirements.

To reduce the time complexity of the search operation on the collector, it is also possible to decompose a huge encrypted index into multiple smaller instances. During the search operation, the collector is able to use an efficient search algorithm, such as binary search, on these instances and retrieve the corresponding log IDs. This construction also enables parallel search on multiple instances of encrypted index through multi-threading development. SELL v1.0 does not provide such a mechanism, but it offers the flexibility of design and development of such an architecture for developers. We plan to incorporate multi-threading search over a distributed encrypted index in the upcoming versions of the library.

Finally, switching to lower levels of security can improve the overall performance of the library significantly.

---

[8] Available at: https://github.com/kiritbasu/Fake-Apache-Log-Generator (Accessed August 2018).

[9] Available at: https://swtch.com/~rsc/regexp/regexp4.html (Accessed on August 2018).

6 Conclusion and Future Work

In this work, we presented a new approach toward a fully secure logging system. We designed and developed a framework that provides confidentiality, integrity, completeness, and non-repudiation of logs. Our library also enables specific operations such as keyword searching and discovering how many times a keyword has occurred on encrypted logs. The proposed method preserves the confidentiality of log data by using an encrypted inverted index that does not leak any information about the logs and keywords to the semi-honest collector and is IND-CKA2 secure. We defined a set of default parameters for the SELL v1.0 that provide a *conservative* standard security level against a variety of attacks.

In terms of performance, we benchmarked SELL v1.0 with a batch of log samples, bootstrapping the system with 10,000 lines of logs and evaluated the timing of each operation on an Intel Core i7 processor. Our results imply that the proposed solution is practically feasible to deploy in real settings, and that it provides a strong security level. Results can be improved notably by setting the parameters at lower security levels or further optimizing the implementation.

We plan to extend the functionality and performance of the library in the next versions. Our main priority in achieving better performance is designing and developing index decomposition and parallel searching. We hope this work inspires researchers and engineers to investigate into the security and performance of secure operations on encrypted logs, providing all the necessary security assurances for the users' data.

References

1. Bellare, M., Rogaway, P.: Introduction to modern cryptography. Ucsd Cse 207, 207 (2005)
2. Chai, Q., Gong, G.: Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In: Communications (ICC), 2012 IEEE International Conference on. pp. 917–922. IEEE (2012)
3. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. Journal of Computer Security 19(5), 895–934 (2011)
4. Dworkin, M.J.: Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Tech. rep. (2007)
5. Goh, E.J., et al.: Secure indexes. IACR Cryptology ePrint Archive 2003, 216 (2003)
6. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 965–976. ACM (2012)
7. Kelly, S., Frankel, S.: RFC 4868-Using HMAC-SHA-256. Tech. rep., HMAC-SHA-384, and HMAC-SHA-512 with IPsec, http://www. ietf. org/rfc/rfc4868. txt (2007)
8. Kelsey, J., Callas, J., Clemm, A.: Signed syslog messages. No. RFC 5848. Tech. rep. (2010)
9. Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: Recent Advances in Intrusion Detection. pp. 9–9 (1999)

10. Kurosawa, K., Ohtaki, Y.: Uc-secure searchable symmetric encryption. In: International Conference on Financial Cryptography and Data Security. pp. 285–298. Springer (2012)
11. Law, L., Solinas, J.: Suite b cryptographic suites for ipsec. Tech. rep. (2007)
12. Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an encrypted and searchable audit log. In: NDSS. vol. 4, pp. 5–6 (2004)