A Security Proxy to Cloud Storage Backends based on an Efficient Wildcard Searchable Encryption

Shen-Ming Chung
Department of Electrical Engineering
National Cheng Kung University
Tainan, Taiwan
anton0706@gmail.com

Ming-Der Shieh
Department of Electrical Engineering
National Cheng Kung University
Tainan, Taiwan
shiehm@mail.ncku.edu.tw

Tzi-Cker Chiueh
Information and Communication Laboratories
Industrial Technology Research Institue
HsinChu, Taiwan
tcc@itri.org.tw

***Abstract*—Cloud storage backends such as Amazon S3 are a potential storage solution to enterprises. However, to couple enterprises with these backends, at least two problems must be solved: first, how to make these semi-trusted backends as secure as on-premises storage; and second, how to selectively retrieve files as easy as on-premises storage. A security proxy can address both the problems by building a local index from keywords in files before encrypting and uploading files to these backends. But, if the local index is built in plaintext, file content is still vulnerable to local malicious staff. Searchable Encryption (SE) can get rid of this vulnerability by making index into ciphertext; however, its known constructions often require modifications to index database, and, to support wildcard queries, they are not efficient at all. In this paper, we present a security proxy that, based on our wildcard SE construction, can securely and efficiently couple enterprises with these backends. In particular, since our SE construction can work directly with existing database systems, it incurs only a little overhead, and when needed, permits the security proxy to run with constantly small storage footprint by readily out-sourcing all built indices to existing cloud databases.**

Keywords—cloud storage, security proxy, searchable encryption

I. INTRODUCTION

Cloud services such as Dropbox have been popular and kept attracting those who are willing to see their files ready for access anytime and anywhere. With the help of *cloud storage backends* such as Amazon S3, these services can focus on provisioning abundant features without worries about the scalability and the availability of the underlying storage. However, despite the convenience and the possible IT cost-down, enterprises could still prefer on-premises storage than these services simply because of confidentiality. It is observable that, besides being a building block to cloud services, cloud storage backends themselves can be a potential storage solution to enterprises if two intrinsic problems can be solved. Namely, comparing to on-premises storage, 1) these backends generally do not support content search and thus will make it clumsy for file retrieval; 2) they are usually considered *semi-trusted* [1] and thus the files stored there is vulnerable to attacks particularly from "insiders" of the backends, even if outside hackers are perfectly blocked.

Addressing similar problems, related works were presented in [2][13][14] that enforce encryption over confidential data before uploading to cloud. However, due to different focuses, each of the works either omits file content search or builds local index in plaintext that makes local malicious staff easy to peek at keywords of all files without authorization. It is known that *index-based Searchable Encryption* (SE) [3][4] can build index in ciphertext so as to prevent eavesdropping while enabling search ability. However, it is also known that SE constructions often require modifications or even a new design to existing databases in order to make such *secure index* working, and, if *wildcard queries* are to be served, known SE constructions such as [5-9] are not efficient for large data volume as they require a search time linear to the number of indexed items at best.

To securely and efficiently couple enterprises with cloud backends, in this paper we propose a security proxy based on our wildcard SE construction. Section II gives the design of the proposed security proxy which incurs only a little overhead by means of our wildcard SE construction depicted in Section III. The evaluation in Section IV affirms the viable performance. Section V concludes this work.

II. SECURITY PROXY

A. Overview of the architecture

The architecture of our security proxy is shown in Fig. 1 that, though simplified, highlights the open-source tools we use and their relation; wherein *Eclipse Jetty* is extended to accept S3 API requests with each Jetty handler serving an incoming S3 request. An *H2 database* is used as a keystore that is filled with clients' proxy/backend credentials for authentication. AES module conducts encryption (decryption) for uploaded (downloaded) files. *Apache Tika* helps extract keywords from uploaded files of a variety of formats while our SE module builds for each unique keyword a secure index. *Apache Solr* is used to store each secure index as a text string that, based on the properties of our SE construction, enables an efficient encrypted search as

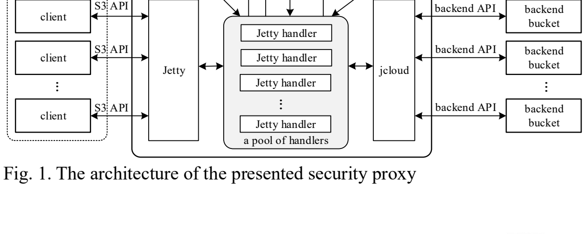


Fig. 1. The architecture of the presented security proxy

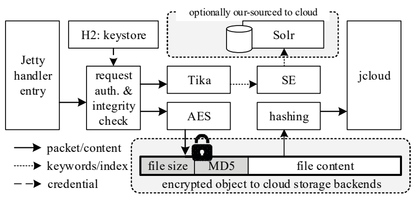


Fig. 2. The flow of uploading process and its outcome

elaborated in section III. To couple with a wide range of cloud storage backends, *Apache jcloud* is used for a unified access.

B. Flow of uploading process

To make the semi-trusted backends as secure as on-premises storage, every file should be encrypted before uploading to backends. When a client is to upload a file to their buckets, a PUT request is made with an IP addressing the security proxy. As shown in Fig. 2, after such a request is authenticated with a right proxy credential and the integrity is confirmed by MD5 (as S3 requires), the keywords of the uploaded file are extracted by Tika, built into secure indices by our SE construction and then stored with file's backend URI as a Solr document for queries. Concurrently, after the file is encrypted by AES using a client-designated key and packed into an object prepended with the original file size and MD5, the object are uploaded to the target backend bucket in a way compliant for backend authentication. The prepended header is a trick for fast downloads.

C. Flow of downloading process

Similarly, when a client is to retrieve a file by an (identified) URI from a certain bucket, a GET request is made with an IP addressing the security proxy. As shown in Fig. 3, after the request is authenticated with integrity, the encrypted object of the URI is retrieved and decrypted on-the-fly. A response to the client is made of a header and a payload. With the help of the prepended original file size and MD5, the response header can be returned to the client instantly without waiting for all bytes of the file transferred from the bucket.

D. Query for identifying wanted file (URIs)

To make file retrieval as efficient as on-premises storage, the security proxy allows clients to first identify wanted file URI(s) by *wildcard queries* before the actual retrieval. By the properties of the secure indices built from our SE construction, this process of encrypted search only requires existing database such as Solr. Remarkably, such properties make all built secure indices easily out-sourced to *Solr as a service* such as SearchStax [10].

III. SE Construction for the Security Proxy

Our security proxy supports wildcard queries for selectively retrieving file(s). For this purpose, each unique keyword per file is built into an SE index and each querying keyword an SE trapdoor. This section presents how each SE index and each SE trapdoor are built and why they work.

Intuitively, like distorting mirrors, our SE construction reflects and projects an object (a given keyword) onto an obfuscating image (SE index); and in case that you cover (wildcardize) a part of the object (the keyword), the part that is

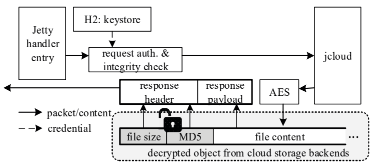


Fig. 3. The flow of downloading process

not covered still gets reflected and projected onto another image (SE trapdoor) which, though still obfuscating, is in part the same as the image without cover.

A. SE Index

As illustrated in Fig. 4, given a keyword $\boldsymbol{p} = [p_1 \quad p_2 \quad \cdots \quad p_N]^{\mathrm{T}}$, there are three steps for building an SE Index, namely, *reflecting*, *projecting* and *garbling*. In the step of reflecting, characters of $\boldsymbol{p}$ are "reflected" by a multiplication with a *secret matrix* $\boldsymbol{R}$ *of M-by-N* entries defined as

$$\boldsymbol{R} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \boldsymbol{r}_3 \\ \vdots \\ \boldsymbol{r}_M \end{bmatrix} \tag{1}$$

with each of $\boldsymbol{r}_{m\in\{1,2,\ldots M\}}$ defined as randomly-shuffled version of $\boldsymbol{z} = [z_1\ z_2\ \ldots\ z_N] \overset{\text{def}}{=} [x^0\ x^1\ \ldots\ x^{\alpha-1}\ 0\ 0 \ldots 0]$, where each non-zero entry is called a *reflecting pointer* and α defines the number of reflecting pointers in each row of $\boldsymbol{R}$.

After the reflecting, M sets of reflected characters are collected in the coefficients of polynomials $\boldsymbol{v}_1(x)$, $\boldsymbol{v}_2(x)$, …, and $\boldsymbol{v}_M(x)$. That is,

$$\boldsymbol{Rp} = \boldsymbol{R}\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_1(x) \\ \boldsymbol{v}_2(x) \\ \boldsymbol{v}_3(x) \\ \vdots \\ \boldsymbol{v}_M(x) \end{bmatrix} \tag{2}$$

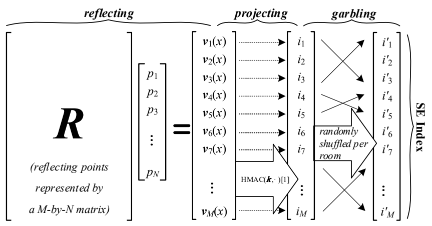
In the step of projecting, each polynomial is projected into a character. One way to do it is by conducting HMAC over the coefficients of each polynomial using a *secret key* $\boldsymbol{k}$ and then selecting the first character from the base64 representation of the resulted hash. Denoting such a projecting as $\succ$, we obtain an *SE image* $\boldsymbol{i} = [i_1 \quad i_2 \quad i_3 \quad \cdots \quad i_M]^{\mathrm{T}}$ as shown below

$$\boldsymbol{i} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{v}_1(x)^{\succ} \\ \boldsymbol{v}_2(x)^{\succ} \\ \boldsymbol{v}_3(x)^{\succ} \\ \vdots \\ \boldsymbol{v}_M(x)^{\succ} \end{bmatrix} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ \vdots \\ i_M \end{bmatrix} \tag{3}$$

For the step of garbling, we partition the SE image $\boldsymbol{i}$ into λ "rooms" by a *secret sequence* $\boldsymbol{b}$ of increasing integers, i.e.

$$\boldsymbol{b} \overset{\text{def}}{=} (b_1, b_2, \ldots, b_{\lambda+1})|_{b_1=1, b_{\lambda+1}=M+1, b_i-b_{i-1}>2} \tag{4}$$

That is,


reflecting
projecting
garbling
R
(reflecting points represented by a M-by-N matrix)
HMAC(k,·)[1]
randomly shuffled per room
SE Index
Fig. 4. Process of building an SE index

$$\boldsymbol{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ \vdots \\ i_M \end{bmatrix} = \begin{bmatrix} \boldsymbol{i}_1 \\ \boldsymbol{i}_2 \\ \vdots \\ \boldsymbol{i}_\lambda \end{bmatrix} \tag{5}$$

such that

$$\boldsymbol{i}_l = \begin{bmatrix} i_{b_l} \\ i_{b_l+1} \\ i_{b_l+2} \\ \vdots \\ i_{b_{(l+1)}-1} \end{bmatrix} \tag{6}$$

Now we introduce a tool called ***Garbler G*** defined as

$$\boldsymbol{G} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{\pi}_1 \\ \boldsymbol{\pi}_2 \\ \boldsymbol{\pi}_3 \\ \vdots \\ \boldsymbol{\pi}_\lambda \end{bmatrix} \tag{7}$$

where each $\boldsymbol{\pi}_{l\in\{1,2,\dots,\lambda\}}$ is a randomly-generated *permutation matrix* with dimension $(b_{l+1} - b_l) \times (b_{l+1} - b_l)$ and with $\boldsymbol{p}$ as the seed to a random source.

With $\boldsymbol{G}$, a *SE Index* $\boldsymbol{idx}$ is obtained by

$$\boldsymbol{idx} \overset{\text{def}}{=} \boldsymbol{G}^{\mathsf{T}}\boldsymbol{i} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{\pi}_1\boldsymbol{i}_1 \\ \boldsymbol{\pi}_2\boldsymbol{i}_2 \\ \boldsymbol{\pi}_3\boldsymbol{i}_3 \\ \vdots \\ \boldsymbol{\pi}_\lambda\boldsymbol{i}_\lambda \end{bmatrix} \tag{8}$$

B. SE Trapdoor for a (wildcardized) keyword

To query for a specific (wildcardized) keyword, an SE trapdoor is to be built by a client knowing the key $\{\boldsymbol{R}, \boldsymbol{k}, \boldsymbol{b}\}$. As illustrated in Fig. 5, given a keyword $\boldsymbol{q} = [q_1 \quad q_2 \quad \cdots \quad q_N]^{\mathsf{T}}$, the process of building SE Trapdoor also comprises three steps. In the step reflecting, *M* polynomials are obtained similarly, i.e.

$$\boldsymbol{Rq} = \boldsymbol{R} \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_1(x) \\ \boldsymbol{v}_2(x) \\ \boldsymbol{v}_3(x) \\ \vdots \\ \boldsymbol{v}_M(x) \end{bmatrix} \tag{9}$$

Denoting a wildcard that represents any character as "?", if $\boldsymbol{q}$ contains "?", "?" could be reflected and collected in the coefficients of $\boldsymbol{v}_1(x)$, $\boldsymbol{v}_2(x)$, …, and $\boldsymbol{v}_M(x)$. For each polynomial containing at least one "?", we force the result of

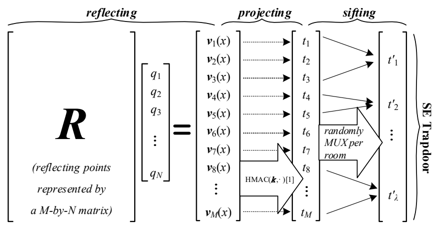


Fig. 5. Process of building an SE trapdoor

projecting $\succ$ to be a "?" too. Therefore, we might obtain a "?"-involved $\boldsymbol{t} = [t_1 \quad t_2 \quad \dots \quad t_M]^{\mathsf{T}}$ that, for the step of *sifting*, is similarly partitioned into λ rooms by the secret sequence $\boldsymbol{b}$, i.e.

$$\boldsymbol{t} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{v}_1(x)^{\succ} \\ \boldsymbol{v}_2(x)^{\succ} \\ \boldsymbol{v}_3(x)^{\succ} \\ \vdots \\ \boldsymbol{v}_M(x)^{\succ} \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_M \end{bmatrix} = \begin{bmatrix} \boldsymbol{t}_1 \\ \boldsymbol{t}_2 \\ \vdots \\ \boldsymbol{t}_\lambda \end{bmatrix} \tag{10}$$

Now we introduce another tool called ***Sift S*** defined as

$$\boldsymbol{S} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{\sigma}_1 \\ \boldsymbol{\sigma}_2 \\ \boldsymbol{\sigma}_3 \\ \vdots \\ \boldsymbol{\sigma}_\lambda \end{bmatrix} \tag{11}$$

wherein each $\boldsymbol{\sigma}_{l\in\{1,2,\dots,\lambda\}}$ is a randomly-generated *selection vector* of dimension $(b_{l+1} - b_l)$ that has exactly one entry of 1 and 0's elsewhere. By $\boldsymbol{\sigma}_{l\in\{1,2,\dots,\lambda\}}$, one can select (i.e. mux) one character per room using inner product with $\boldsymbol{t}_{l\in\{1,2,\dots,\lambda}$ and get a *SE Trapdoor* $\boldsymbol{trap}$ as

$$\boldsymbol{trap} \overset{\text{def}}{=} \boldsymbol{S} \cdot \boldsymbol{t} \overset{\text{def}}{=} \begin{bmatrix} \boldsymbol{\sigma}_1 \cdot \boldsymbol{t}_1 \\ \boldsymbol{\sigma}_2 \cdot \boldsymbol{t}_2 \\ \boldsymbol{\sigma}_3 \cdot \boldsymbol{t}_3 \\ \vdots \\ \boldsymbol{\sigma}_\lambda \cdot \boldsymbol{t}_\lambda \end{bmatrix} \tag{12}$$

C. Application Notes

Due to limited space, proofs and security analysis are skipped but some application notes are given for the properties of the presented SE construction:

1) A recall rate of 1: It can be verified that under the same key $\{\boldsymbol{R}, \boldsymbol{k}, \boldsymbol{b}\}$, given $\boldsymbol{q} = \boldsymbol{p}$, the $\boldsymbol{trap}$ built for $\boldsymbol{q}$ must be a subsequence of $\boldsymbol{idx}$ built for $\boldsymbol{p}$; and if some characters of the $\boldsymbol{q}$ are replaced by "?", the $\boldsymbol{trap}$ built for such a $\boldsymbol{q}$ must still be a subsequence of $\boldsymbol{idx}$ built for $\boldsymbol{p}$ as "?" represents any character. This makes the SE construction a recall rate of 1 that requires only a database supporting *subsequence searching*, such as Solr.

2) Shorter keywords: Defining L as a *threshold*, keywords (queries) with length $l < L < N$ can fit with the SE by first padding with $(L - l)$ characters from $HMAC(\boldsymbol{k}, l)$, then the padded word is XORed with other characters from $HMAC(\boldsymbol{k}, l)$ and repeatly concatenated until the length reaches N.

3) Query Privacy: The presented SE obfuscates *search pattern* [4] in order to make it hard to track if any (wildcard) query ever repeats and thus enhances the protection on privacy.

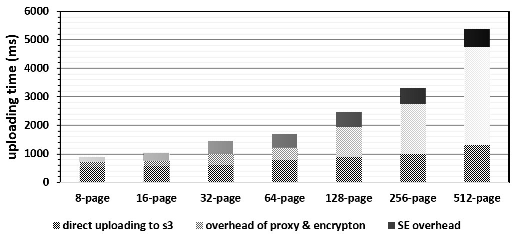


Fig. 6. Overheads of the security proxy to an S3 bucket in N. Virginia

4) Security: Our SE sacrifices *index indistinguishability* [3] for search speed; but, with the help of the garbling and sifting, it manages to prevent character frequency analysis attacks by mixing up the frequency traits leaked from HMAC.

IV. Performance Evaluation

A computer with an Intel i7-7700 CPU running at 3.6GHz is used for running both the security proxy and a client that uploads files. An SE instance of $N = 40$, $M = 200$, $\alpha = 7$, $\lambda = 40$ and $L = 16$ is used and can be disabled for measuring its overhead.

A. *Overhead of the security proxy*

Since the security proxy has to parse the textual content of uploaded files, the overheads of the security proxy rather depend on the textual volume than file size. Thus, to evaluate overhead per textual volume, a PDF of a textbook [11] is used as a baseline and segmented into copies of the first 8-page, 16-page, …, and 512-page. For each copy, uploading time is measured for three cases: 1) direct to S3, 2) via proxy with encryption, and 3) via proxy with encryption and SE enabled. With 1000 tests for each case, estimated overheads are shown in Fig. 6. Note that, SE overhead is bounded by the number of unique keywords per file.

B. *Search Time*

To measure the search time for large data volumes, Enron dataset [12] is used. As shown in Fig. 7, where each measured point is an average of the results of 1000 queries randomly picked from indexed keywords, the SE instance gives excellent search speed upon Solr to all queries with zero to 4 consecutive wildcards (that simulates the wildcard $*$ that represents arbitrary number of characters) done within 440ms. Note that, due to the randomness of $\boldsymbol{R}$, using wildcards in consecutive order or not does not statistically affect the result.

C. *Search Precision*

Precisions, i.e. the ratio between correctly matched items among all identified, are measured for queries with zero to 4 wildcards. As shown in Fig. 8, the precisions of queries with up to three consecutive wildcards are very close to 1 while the precision of queries with four wildcards drops substantially when thousands of documents are indexed; such a drop is due to too-short SE trapdoors built from keywords masked by four wildcards of "? " that falsely identify other SE indices.

V. Conclusion

This paper proposes a security proxy for enterprises to couple with cloud storage backends for benefits. Architecture and processing flows are detailed for demonstrating how the proxy works for multiple clients and backends. To enhance security, a wildcard SE construction is presented that, by transforming encrypted search into the problem of subsequence search, well fits with the architecture and incurs only a little overhead without the need to modify the database system. It can be observed that, besides enterprises, our security proxy can also readily enhance the security of general cloud services that rely on cloud storage backends.

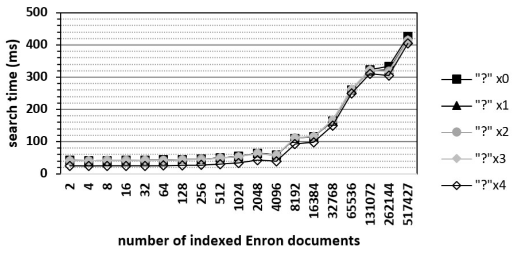


Fig. 7. Search time over secure indices built by the SE instance

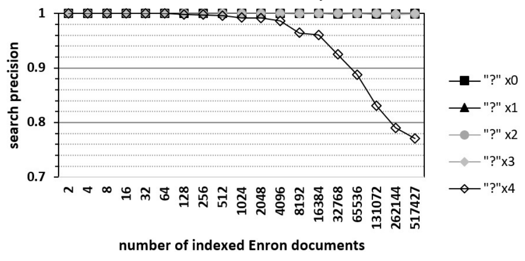


Fig. 8. Search precision over wildcard queries by the SE instance

References

[1] Oded Goldreich, "The Foundations of Cryptography - Volume 2, Basic Applications," Cambridge University Press, 2004

[2] C. Liu, G. Wang, and et al, "A cloud access security broker based approach for encrypted data search and sharing," In Proc. of ICNC 2017, pp. 422-426

[3] E. J. Goh, "Secure indexes," Cryptology ePrint Archive, 2004

[4] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. "Searchable symmetric encryption: Improved definitions and efficient constructions," In CCS. ACM, New York, NY, 79–88. 2006

[5] S. Sedghi, P. V. Liesdonk, S. Nikova, P. H. Hartel and W. Jonker. "Searching keywords with wildcards on encrypted data," In SCN (LNCS), Vol. 6280. Springer, 138–153, 2010

[6] C. Bösch, R. Brinkman, P. Hartel, W. Jonker, "Conjunctive wildcard search over encrypted data," In Proc. of SDM 2011, pp. 114–117

[7] C. Hu, L. Han. "Efficient wildcard search over encrypted data," International Journal of Information Security, 1-9, 2015

[8] D. Wang, X. Jia, and et al, "Generalized pattern matching string search on encrypted data in cloud systems," In Proc. of INFOCOM 2015, pages 2101–2109, April 2015.

[9] Y. Yang, X. Liu, R. H. Deng, and J. Weng, "Flexible wildcard searchable encryption system," IEEE Transactions on Services Computing, 2017.

[10] SearchStax, https://www.searchstax.com/

[11] D. Boneh and V. Shoup, "A Graduate Course in Applied Cryptography," https://crypto.stanford.edu/~dabo/cryptobook/, Version 0.4

[12] Enron dataset, https://www.cs.cmu.edu/~./enron/

[13] M. Vrable, S. Savage, and G. M. Voelker. "BlueSky: A Cloud-backed File System for the Enterprise." In Proc. of the USENIX Conference on File and Storage Technologies (FAST), 2012.

[14] R. Zhao, C. Yue, B. Tak, C. Tang, "SafeSky: A Secure Cloud Storage Middleware for End-User Applications", IEEE 34th Symposium on Reliable Distributed Systems (SRDS), 2015.