# Multi-user Searchable Attribute Based Encryption for Outsourced Big Data

Kyle Astudillo
*Computer Science Department*
*CSU Northridge*
Northridge, United States
kyle.astudillo.965@my.csun.edu

Mahdi Ebrahimi
*Computer Science Department*
*CSU Northridge*
Northridge, United States
mahdi.ebrahimi@csun.edu

Adam Kaplan
*Computer Science Department*
*CSU Northridge*
Northridge, United States
adam.kaplan@csun.edu

***Abstract*—Petabytes of data are generated every minute, and this creates a corresponding demand for servers which ingest, process, store, and maintain this data. The scale of data has become sufficiently large that normal file access and processing present different challenges than in prior decades. In the Web 2.0 era, data owners commonly stored and shared data using on-premise servers. In the past decade, cloud computing has quickly become the dominant hosting paradigm, as it offers data owners the benefits of cost-savings, flexible scaling, and ease of administration. However, cloud computing presents a myriad of security challenges. Attribute based encryption has been key to tackling these challenges. We present a method to allow users to maintain their own secure data on a shared system, while eliminating the time penalty and single point-of-failure associated with traditional authorization techniques. This is implemented as a Representational State Transfer API, removing the burden of managing auxiliary files or encryption libraries when uploading documents. This system also supports data encrypted at rest, removing concerns of administrator access to unencrypted contents in storage. We leverage key policy attribute-based encryption (KP-ABE) to store this data, search its contents, and receive/decrypt this data. We further demonstrate that KP-ABE can be implemented as part of a distributed or centralized authorization method. We compare KP-ABE against AES cryptography, and present system metrics from its execution on representative data.**

***Index Terms*—Attribute based encryption, cloud storage, REST API, Key policy attribute based encryption, encryption at rest**

**Submission Type:** Full/Regular Research Paper
**Symposium:** CSCI-RTBD

## I. Introduction

In the 2010s, cloud-hosting became a popular option upon which to deploy organizational infrastructure and file storage. When the cloud is combined with consumption-based billing, service abstraction, and scalability, cloud computing can offer compelling value to organizations deploying web services. Sedayao et al highlight major incentives of cloud hosting, including cost-savings, flexible scaling, and ease of administration [1]. However, a major barrier blocking enterprise adoption of cloud computing is the associated security concern. Organizations must give careful consideration to the task of retaining data confidentiality. Sedayao et al bring to light problems that are inherent in cloud hosting, including the potential of data which is not encrypted at rest. If data resides in storage in unencrypted plaintext form, a cloud service administrator can simply read the data directly from storage. This problem is compounded when organizations want to share files with business partners, accounting firms, or law firms. In these situations, encrypted files leave a trusted system and have to be transferred or copied into untrusted systems. With organizations now incentivized to reap the benefits of cloud pricing, there is an increased need for a suitable security technique that addresses these issues without negatively impacting application performance nor other hosting metrics.

Attribute based encryption (ABE) has been identified as an effective encryption strategy for cloud hosting. ABE encrypts data at rest, allowing data to hide in plain sight in the event of an unauthorized storage read-access. When sharing data with another organization ABE enforces an authorization method within the decryption key itself allowing file-decryption only by users with the correct key and attributes satisfying its policies [2]. OpenABE, an open source implementation of attribute based encryption, popularly describes an example Key Policy Attribute Based Encryption (KP-ABE) of email data, where attributes of this data include the header fields from, to, and date [8]. In this example, a data owner sharing multiple encrypted emails can generate a key which only decrypts emails from or to specific users, sent within a given date range [8].

In classic cryptography, this scenario introduces a problem where users possess the encrypted data, but now must decrypt each file to see which files they have access to. Morales-Sandoval et al proposed a solution using ABE which encrypts incoming data by extracting keywords from the original documents [3]. This allows end users to search encrypted documents with an encrypted search query. This method is named Attribute Based Searchable Encryption (ABSE). The system proposed by Morales-Sandoval et al makes it possible to share encrypted data with many users, each able to search/access this data using KP-ABE keys.

In this paper, we present an ABE system providing storage, sharing and retrieval of encrypted data. This data is encrypted end-to-end and at rest. We extend prior art by giving data owners the option of allowing multiple users to store encrypted data in the system. This provides a many-to-many relationship between data producers to data consumers. We describe a Representational State Transfer (RESTful) API to interact with

979-8-3503-2028-2/22/$31.00 ©2022 IEEE
DOI 10.1109/CSCI58124.2022.00212

the system. This eases the uploading and downloading of files, removing the burden of managing local encryption software or keys on user systems. This system is thus cross-compatible with multiple operating systems, as OpenABE is currently a cross-compatible library. The system provides for users to maintain the encryption software on their own system so that they may upload files into the system encrypted with their own extracted keywords.

The remainder of this paper is organized as follows. In Section II, we present prior art in ABE cryptography. We describe our system implementation in Section III. In Section IV we describe the create, update, read, and delete operations as implemented in this system, and in Section V we describe our experimental methodology. We present experimental results in Section VI, and conclude in Section VII.

## II. Related Work

Bethencourt et al provided the first construction of a ciphertext-policy attribute-based encryption (CP-ABE) [4]. ABE differentiates itself from other encryption techniques by embedding access control into cipher-text [2]. This ability to embed access control information into the file itself allows for the ability of checking permissions on the client side during the decryption phase, without the need for a centralized identity and access management system. Any user who has an identity that contains all of the necessary attributes could decrypt the document [6]. This provides cost-effective storage and retrieval of encrypted data in the cloud, while distributing data access security at scale.

OpenABE provides three types of policies of encryption public-key (PKE), ciphertext-policy (CP-ABE), and key-policy (KP-ABE) [8]. CP-ABE and KP-ABE are variants of ABE which handle policies and attributes differently. CP-ABE stores policy trees in the ciphertext and attributes with the users [4], while KP-ABE stores policy trees with the users and attributes with ciphertext [5]. As an example of CP-ABE behavior, we consider the case of a user described by name, job title, age, or rank. In CP-ABE, a stored file could determine who can gain access to the content by specifying a job title or a minimum qualifying rank [8]. An example of KP-ABE behavior is the inverse, wherein attributes describing the file are used, and the user is granted a policy of their accessible files by describing a date or content type [8]. The present paper makes use of KP-ABE, and investigation of CP-ABE is left for future work.

The KP-ABE process to share data is as follows. In the setup phase, a master secret key (MSK) and public parameter store (MPK) are generated [8]. Next, using the MSK and MPK, a key can be generated. The key will have a policy tree that describes what kind of content may be accessed, and simply needs to be given to anyone with whom the data owner wants to share corresponding data. To encrypt data in a KP-ABE system an MPK, a random number, attributes, and a content file are needed to create an encrypted file. To decrypt data in a KP-ABE system an MPK, a key, and encrypted file are used. If the policy, attributes, and parameter store align, then the content will be decrypted. Otherwise the decryption process will not succeed.

Morales-Sandoval et al extended these ABE processes so that a data owner could not only encrypt files but store them in the cloud though a cloud service provider (CSP) [3]. The CSP indexes files by extracting keywords with which a data consumer could search over file contents. This process is called key-policy attribute based searchable encryption (KP-ABSE), and is elaborated in Fig. 1.

Fig. 1 shows the process with which a data owner would extract words from plaintext data to create a secure index, which can then be used by an encrypted document search node. Those same documents are encrypted into ciphertext and placed into a database hosted by a CSP. This provides data consumers with the ability to create an encrypted query to search through documents, allowing them to then retrieve specific plaintext documents to which they have access (through their attribute set).

We contribute to this body of work by providing a many-to-many relationship between data producers and data consumers. We further demonstrate that KP-ABE can be implemented as part of a distributed or centralized authorization method. We compare KP-ABE against advanced encryption standard (AES) cryptography [9]. We present system metrics from its execution on representative data.

## III. System Model

We implement our system as the following set of functions, hosted on a web server as a RESTful API.

**Word Set Extraction**: from an input plaintext file, words are extracted so that search queries can be made against the words. This function includes storing the file within the private database. As an option this process can be bypassed if the request provides alternate keywords in the JSON payload, which can be used instead of those extracted by the function.

**Secure Index ID (SIID) Creation**: a public webserver requests a SIID from a private webserver for a plaintext file to be saved within the system.

**Secure Index ID To File**: the webserver sends an SIID to the public database to receive an s3-cipher-text-key.

**Event Return Code URL**: provides a response with information such as a status-code, message, stdout, stderr and an optional text data within the content payload. With this, data consumers may poll for results on an alternate URL if the present query takes too long to compile the requested file list.

**Document Encryption**: the webserver makes a call to key-operations wrapper with information such as organization and the key with which to encrypt. This function returns a ciphertext file. The encryption process can be bypassed by users whose files are already encrypted by setting the kpabe-encrypted parameter to true.

**Upload Document**: the plaintext file is uploaded to the private object store (an AWS S3 Bucket) and the ciphertext file is uploaded to the public object store (also an AWS S3 Bucket).

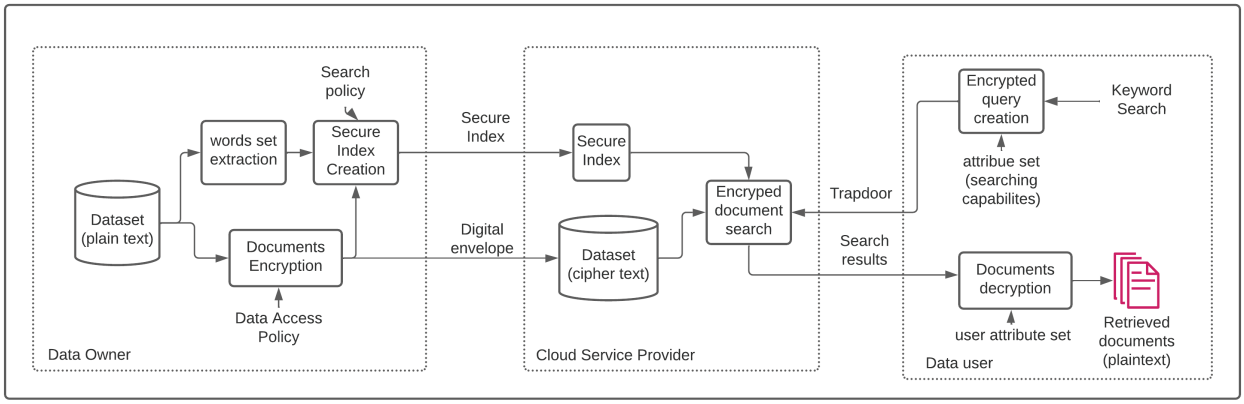


Fig. 1. Morales-Sandoval's System Model for Document Sharing and Retrieval in the Cloud [3].

**Update Event**: the webserver updates all of the databases with the response status. This function stores the object store keys in the private database and the s3-cipher-text-key, status-code, and message in the public database.

**Delete Document**: given an SIID, will remove the SIID from the system which means users will no longer be able to obtain corresponding files though the webserver.

**Encrypted Document**: the webserver will query the private database for a list of SIIDs that match a given search query.

**Get Document**: the webserver retrieves a document from object store to return to the user.

**Document Decryption**: given a set of attributes, and a decryption key, a corresponding plaintext file will be returned.

## IV. Data Operations

The Create operation, shown in Fig. 2, is one of the most resource-intensive operations in the system. Create takes a content file from a data producer and stores the file on the cloud so that data consumers may access the file. The process involves a client making a request to the web server with a file with content, a mpk file to use for encryption, and a config file in JSON with parameters such as organization and policy name. On the webserver once the request is received the word set extraction process begins, in which the file is stored for search queries. Next the webserver makes a request to the CSP database for a secure index identifier. Then a response is returned to the user with a response-id that they can query to see the status of their upload. A separate internal request is made to the key operations server to encrypt the content file which will return an encrypted ciphertext file back to the webserver. With the ciphertext file the webserver can now start the upload process which then stores the plaintext file into an AWS S3 bucket which returns a s3-plain-text-key and stores the encrypted file into an S3 bucket which returns a s3-cipher-text-key. With both files saved an update event takes place which stores s3-plain-text-key, s3-cipher-text-key, ssid, msk, mpk, status-code. The user has the option to poll the status of

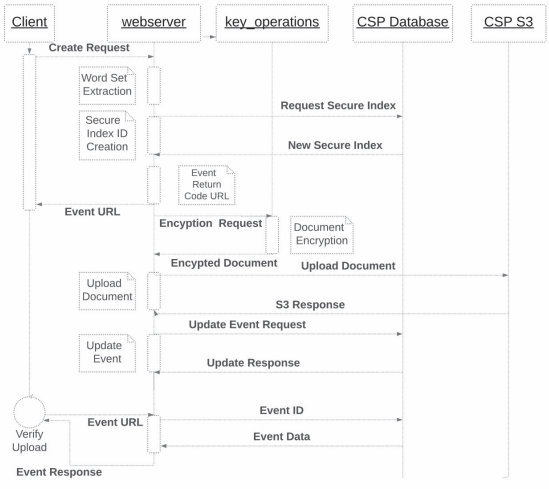


Fig. 2. Create Sequence Diagram

their previous request. The status-code "complete" indicates that the data has been uploaded successfully.

Read operations are one of the more time-consuming operation within the system. Data consumers have access to this operation, which consists of taking a search query and returning a list of files for the client to download. The "Read" operation is more of a search engine while the "Read Id" is a more traditional file-read. The read request is sent from a data consumer client to the webserver with a search query as an attribute in a JSON payload. The webserver then performs an encrypted document search which will return a list of SIIDs for the client to use inside a "Read ID" request.

The "Read Id" operation, shown in Fig. 3, starts by sending a request to the webserver with an SSID as a parameter alongside an mpk file. The webserver then queries the CSP database to obtain the s3-cipher-text-key. The webserver then uses the s3-cipher-text-key to access the ciphertext file from

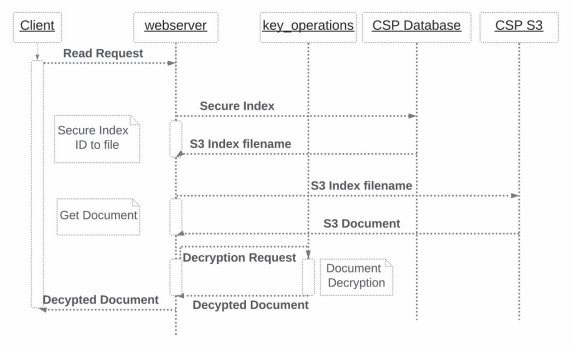


Fig. 3. Read ID Sequence Diagram

an AWS S3 bucket. The returned ciphertext file is sent to key_operations, which decrypts the file and returns a plaintext file which can be returned to the client.

The Update operation is the most complex of the operations. Data producers are authorized to modify existing files within the system. Data producers begin with a request much like the Create operation such that they need a content file, mpk file, and config file with settings such as organization, policy name, SSID, and optional keywords. The webserver uses this to perform a word set extraction on the content file. An SIID is then generated and a status code URL is returned as a response to the data producer. The data producer may poll the URL to see the status of the update operation in progress. The webserver then takes the content file and sends it to key_operations with the mpk file to encrypt the document. Next key_operations returns an encrypted ciphertext file to the web server. Then the web server begins the upload process where the plaintext and ciphertext files are uploaded to their respective S3 buckets. Next an update event is triggered to update the database with the new SIID and s3-cipher-text-key, s3-public-text-key, ssid, msk, mpk, status-code. After this is complete, the status code URL will indicate ”success.”

All create, update, read, delete operations have similar error-handling. If any operation fails before the Event URL is sent, an HTTP error will be returned directly to the user. However if the Event URL has already been sent to the user, the error message will be saved into the response table. Any subsequent request to the Event URL will provide an error message detailing the specific failure.

## V. Methodology

OpenABE was deployed on a virtual machine running CentOS-8-Vagrant-9.3.2011-20201204.2.x8664, 1 Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 16MB of RAM, 16 GB of SSD Memory. This virtual machine was used to run all the experiments natively without any network connections to reduce any variation between test executions. Each test was executed for 1000 iterations to obtain a reasonable sample size for the experiments. All commands were measured with the UNIX time command. The following are the experiments: abe-base-enc, abe-base-dec, aes-base-enc, aes-base-dec, abe-bin-enc, abe-bin-dec, abe-lorem-enc, abe-lorem-dec. All tests have a very similar execution structure: generate a file, KP-ABE encrypt, KP-ABE decrypt, validate output file against generated file. Execution time of each step is measured.

The first set of tests, abe-base-enc and -dec, exercises ABE encryption of a fully-utilized text file containing random characters from the start of the file to end of the file. These characters fully utilize all 16 MB with no spaces. The second set of tests, abe-bin-enc and abe-bin-dec, measures the KP-ABE encryption of 16 MB binary files. The third set of tests, abe-lorem-enc and -dec, exercises KP-ABE encryption of a fully- utilized text file with random words from the start of the file to end utilizing all 16 MB and allowing space characters. The fourth set of tests, aes-base-enc and -dec, measures AES encryption of a fully-utilized text file with random characters from the start of the file to end, fully utilizing all 16 MB with no spaces.

The next set of tests introduces network API calls. These tests no longer take place on a closed virtual machine without network connection. These tests instead use the previously mentioned centos-8 virtual machine, SQL Server version 8.0.31-0, ubuntu 0.20.04.1 for linux on x86-64, and an Ubuntu virtual machine with 1 Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 4 GB of RAM, and 32 GB of SSD Memory. These add some variability to the performance results. However, each time a call outside of the system was made, a measurement was taken before and after. Thus, in the data, each network call will be shown as part of the total time. These tests also no longer use the time command line utility and instead use the python time module to measure elapsed time. These tests were also executed 1000 times to obtain a reasonable sample size.

The fifth set of tests measures how the overall system uploads data and returns data back to users. The test labeled webserver-read measures the execution time of the read operation and the processes behind them such as the download from the public S3 bucket, the decryption process, and the rest of the processes such as getting an s3-cipher-text-key from SIID. These results are grouped together under AVGProcessTime.

## VI. Results

Results from the first four sets of experiments can be seen in Table I. There is some variation between the three file types being encrypted, with the fastest being the encryption of binary text files. The next best file type in terms of speed was the abe-lorem-enc. One of the main contributions to why abe-bin-enc is not up to par with the other two tests is that there was no spaces between the 16 MB of characters, leaving little opportunity for processing at word boundaries. Similarly, for decryption abe-bin-dec files execute fastest, followed by abe-lorem-dec, then abe-base-dec. Encryption is faster by a factor of 17.7, when the mean from or abe-lorem-enc is compared to that of abe-lorem-dec. The decryption process for KP-ABE is a lengthy process with much variability. For example, the best

| Test | USR(s) | SYS(s) | Mem(KB) |
|---|---|---|---|
| abe-base-enc.Mean | 0.13 | 0.08 | 127113.32 |
| abe-base-enc.Median | 0.13 | 0.08 | 127084.00 |
| abe-base-enc.Mode | 0.10 | 0.08 | 127232.00 |
| abe-base-enc.Max | 0.30 | 0.15 | 127292.00 |
| abe-base-enc.Min | 0.09 | 0.05 | 126944.00 |
| abe-base-dec.Mean | 2.05 | 0.07 | 103424.99 |
| abe-base-dec.Median | 2.06 | 0.08 | 103460.00 |
| abe-base-dec.Mode | 1.90 | 0.08 | 103208.00 |
| abe-base-dec.Max | 3.56 | 0.18 | 103728.00 |
| abe-base-dec.Min | 1.88 | 0.04 | 103120.00 |
| abe-bin-enc.Mean | 0.10 | 0.06 | 127083.26 |
| abe-bin-enc.Median | 0.10 | 0.06 | 127060.00 |
| abe-bin-enc.Mode | 0.10 | 0.06 | 127020.00 |
| abe-bin-enc.Max | 0.16 | 0.11 | 127292.00 |
| abe-bin-enc.Min | 0.08 | 0.04 | 124264.00 |
| abe-bin-dec.Mean | 1.91 | 0.06 | 103405.66 |
| abe-bin-dec.Median | 1.90 | 0.06 | 103440.00 |
| abe-bin-dec.Mode | 1.90 | 0.06 | 103476.00 |
| abe-bin-dec.Max | 2.17 | 0.10 | 103724.00 |
| abe-bin-dce.Min | 1.88 | 0.04 | 103052.00 |
| abe-lorem-enc.Mean | 0.11 | 0.07 | 127110.56 |
| abe-lorem-enc.Median | 0.11 | 0.07 | 127064.00 |
| abe-lorem-enc.Mode | 0.11 | 0.06 | 127036.00 |
| abe-lorem-enc.Max | 0.22 | 0.13 | 127292.00 |
| abe-lorem-enc.Min | 0.09 | 0.05 | 126940.00 |
| abe-lorem-dec.Mean | 1.95 | 0.06 | 103422.17 |
| abe-lorem-dec.Median | 1.92 | 0.06 | 103460.00 |
| abe-lorem-dec.Mode | 1.90 | 0.06 | 103208.00 |
| abe-lorem-dec.Max | 2.53 | 0.11 | 103728.00 |
| abe-lorem-dec.Min | 1.88 | 0.04 | 103116.00 |
| aes-base-enc.MEAN | 0.03 | 0.01 | 4916 |
| aes-base-enc.MEDIAN | 0.03 | 0.01 | 4920 |
| aes-base-enc.MODE | 0.03 | 0.01 | 4976 |
| aes-base-enc.MAX | 0.04 | 0.02 | 8176 |
| aes-base-enc.MIN | 0 | 0 | 4760 |
| aes-base-dec.MEAN | 0.06 | 0.02 | 0.01 |
| aes-base-dec.MEDIAN | 0.05 | 0.02 | 0.01 |
| aes-base-dec.MODE | 0.05 | 0.02 | 0.01 |
| aes-base-dec.MAX | 0.15 | 0.03 | 0.03 |
| aes-base-dec.MIN | 0 | 0 | 0 |

TABLE I
DATA SUMMARY FOR ABE-TESTS

| | Total Time | Process Time | S3 Download |
|---|---|---|---|
| MEAN | 8.10002 | 4.47837 | 0.16 |
| MEDIAN | 8.295 | 4.685 | 0.15 |
| MODE | 7.75 | 4.11 | 0.14 |
| MAX | 15.11 | 11.53 | 2.13 |
| MIN | 3.88 | 0.36 | 0.10 |

TABLE II
DATA SUMMARY FOR WEBSERVER-READ

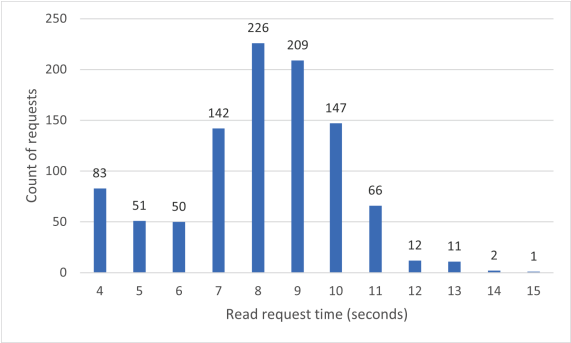


Fig. 4. Read request count graph

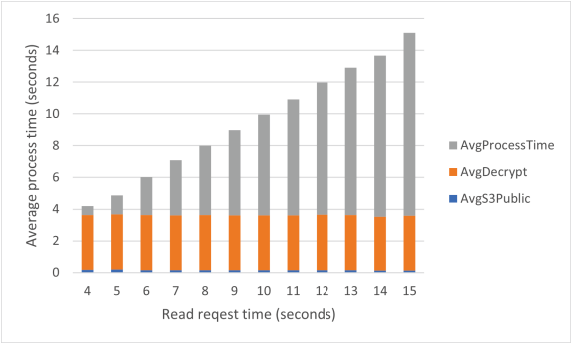


Fig. 5. Read process time graph

case decryption (Min) for abe-lorem-dec is 1.88 user seconds while the worst case (Max) is 2.53 user seconds.

The fourth experiment uses OpenSSL AES [9], a long-popular symmetric encryption algorithm, to compare against OpenABE KP-ABE. The results are 0.03 seconds as an average for encryption and 0.05 seconds for decryption. We compare this result against the average abe-base-enc of 0.13 seconds and average abe-base-dec of 2.05 seconds. This results in AES being 4.3 times faster for encryption and 41 times faster for decryption. The difference between memory usage between these algorithms demonstrates that AES encryption is 25 times more memory-efficient and AES decryption is 10K times more efficient. It should be noted that AES is a symmetric encryption algorithm while ABE is attribute-based encryption, and thus has additional layers built into it. ABE even makes use of AES within its key encapsulation mechanism [7]. As aforementioned, the tradeoff of ABE's memory usage and execution-time is the ability to distribute the user authorization process, thus removing system bottlenecks and single point of failure associated with a classic AES based implementation.

The fifth experiment, summarized in Table II, demonstrates how the system at large performs when saving files into the system, and also when retrieving files. Reading a file from the system can take at least 3.88 seconds and at most 15.11 seconds with 15 MB files. The creation of a file takes longer than a read with 17 seconds for a single 15 MB file upload, however the create operation time within the request is much faster at 0.80 seconds, indicating that the majority (95%) of the create time is spent uploading files to S3 buckets. We find that 80% of create requests complete between 14 and 17 seconds. As Fig. 4 shows, read requests closely approximate a bell curve distribution shifted to the left, with a median time of 8.30 seconds, a minimum time of 3.88 seconds, and a maximum time of 15.11 seconds. The overall execution of the read request takes 8.10 seconds on average but the process time takes 4.47 seconds, comprising 55% of the request's total time. The read decryption time takes 1.95 seconds, which comprises 24% of the overall request total time. The S3 download portion of the read request is only 1% of the total

read operation, using only 0.16 seconds on average with a minimum of 0.10 seconds and a maximum of 2.13 seconds. The distribution of execution time within the read process is shown in Fig. 5. This demonstrates that across read request times, the S3 download portion and the decryption time remain roughly constant, whereas longer-executing reads have longer AvgProcessTime.

## VII. Conclusion

The work described herein maintains secure data at scale while eliminating the time penalty and single point of failure of authorizing users for data access. We implement a distributed security approach using KP-ABE. We leverage the properties of ABE to authorize data users by their attributes, which must satisfy the user policy tree embedded within the decryption key. We extend the system theorized by Morales-Sandoval et al [3] to allow multiple data processing clients to search encrypted data using extracted keywords. The execution burden of authorization is distributed onto each of the clients instead of centralized at the server. We find that the S3 upload of a file dominates (95%) the execution of file creation, but that the S3 download operation comprises only 2% of file reads. We also find that classic AES is 4.3 times faster than KP-ABE for encryption and 41 times faster for decryption. However, ABE provides for more flexible reading and searching of encrypted data at scale, and can eliminate the single point of failure experienced by traditional file encryption systems.

In future work we intend to address key-revocation, which is not currently implemented in this system. This will allow a user whose key has been compromised or whose privileges have been revoked to lose access to data associated with their key. We propose to extend OpenABE to include a use-limit attribute that increments on every decryption. Once the use-limit is reached, a user will need to request a new key from the system, which limits the extent of their authorization. We will implement and measure authentication methods to identify users that have lost their permissions, and deny them service.

## References


[1] J. Sedayao, S. Su, X. Ma, M. Jiang, K. Miao, "A Simple Technique for Securing Data at Rest Stored in a Computing Cloud," IEEE International Conference on Cloud Computing, pp. 553-558, 2009.

[2] M. Green, B. Waters, S. H. Waters, J. Akinyele, The OpenABE Design Document, Zeutro LLC, 2018.

[3] M. Morales-Sandoval, M. H. Cabello, H. M. Marin-Castro, J. L. G. Compean, "Attribute-Based Encryption Approach for Storage, Sharing and Retrieval of Encrypted Data in the Cloud," IEEE Access, 8, 170101-170116, 2020.

[4] J. Bethencourt, A. Sahai, B. Waters, "Ciphertext-policy attribute-based encryption," Proceedings of the 2007 IEEE Symposium on Security and Privacy, pp. 321-324, 2007.

[5] V. Goyal, O. Pandey, A. Sahai, B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 89-98, 2006.

[6] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in Annual International Conference On The Theory And Applications Of Cryptographic Techniques, pp. 457-473, 2005.

[7] J. Akinyele, libopenabe-v1.0.0-api-doc, Zeutro LLC, 2022.

[8] J. Akinyele, libopenabe-v1.0.0-cli-doc, Zeutro LLC, 2022.

[9] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," 1999.