FileCrypt: Transparent and Scalable Protection of Sensitive Data in Browser-based Cloud Storage

Peiyi Han*, Chuanyi Liu†¶, Yingfei Dong‡, Hezhong Pan*, QiYang Song§, Binxing Fang†
*Beijing University of Posts and Telecommunications, Beijing, China
†Harbin Institute of Technology (Shenzhen), Shenzhen, China
‡University of Hawaii, Hawaii, USA
§Tsinghua University, Beijing, China
¶Correspondence to: cy-liu04@mails.tsinghua.edu.cn

***Abstract*—While cloud storage has become a common practice for more and more organizations, many severe cloud data breaches in recent years show that protecting sensitive data in the cloud is still a challenging problem. Although various mitigation techniques have been proposed, they are not scalable for large scale enterprise users with strict security requirements or often depend on error-prone human interventions. To address these issues, we propose FileCrypt, a generic proxy-based technique for enterprise users to automatically secure sensitive files in browser-based cloud storage. To the best of our knowledge, FileCrypt is the first attempt towards transparent and fully automated file encryption for browser-based cloud storage services. More importantly, it does not require active cooperations from cloud providers or modifications of existing cloud applications. By instrumenting mandatory file-related JavaScript APIs in browsers, FileCrypt can naturally support new cloud storage services and guarantee the file encryption cannot be bypassed. We have evaluated the efficacy of FileCrypt on a number of popular real-world cloud storage services. The results show that it can protect files on the public cloud with relatively low overheads.**

I. Introduction

More and more organizations now adopt cloud storage (e.g. *Dropbox*, *Box*) and enterprise cloud services with file storage (e.g. *Gmail*, *Salesforces*) as an essential part of their business. While enterprises can lower their storage ownership costs and support better mobility, data security remains serious concerns. We have seen many headlines regarding cloud storage (or cloud data sharing) breaches. Many major cloud storage service providers such as *Google Drive*, *Dropbox*, and their customers have already been the victims of such attacks [1]–[3]. Meanwhile, cloud providers have incentives to look into, mine, and sell sensitive user data or files.

Skyhigh Network analyzed the cloud usage of 18 million users and found that an average company uses about 923 cloud services, most of which have browser-based cloud storage capabilities. Particularly, 21% of uploaded files in cloud-based file sharing services contain sensitive data, such as intellectual property and trade secrets [4]. To protect sensitive data on browser-based cloud storage, a promising solution is to encrypt files before sending them to the service providers. However, in practice, applying an encryption solution to browser-based cloud storage is still very challenging. In particular, we need to: 1) maintain good user experience while preserving application functionality; 2) provide scalability for various applications. Consequently, we have to achieve the following three requirements:

1. **Security.** An ideal solution should ensure that any file data uploaded into cloud storage must be encrypted before leaving an enterprise's premise. While using direct solutions such as end-to-end encryption (E2EE) may solve the problem, these solutions often require cloud providers to modify applications. Moreover, we can not completely trust cloud providers to provide E2EE. Therefore, we must investigate new solutions.
2. **Usability.** The solution must be easy to use and preserve common user experience and application functionality. Traditional solutions (such as PGP) force users to encrypt files before uploading them into the cloud. Although it is effective, enterprise users are unlikely to adopt it due to the overheads of switching between applications for daily tasks. A user should be able to interact with an application as usual without having to change its normal process. More importantly, the solution should not affect normal application functionalities such as search, which may be affected by encryption. For example, Security Overlay [5], [6] does not meet this requirement because a file is uploaded in the overlay (instead of the original file uploading functionality), which makes the cloud application as a "dumb" storage and degrades its mass storage and search capabilities. In addition, because the data integrity is usually verified at the cloud storage, we must ensure that encryption does not cause verification failures and file-uploading failures.
3. **Scalability.** A proposed solution must be easy to maintain and scalable. It should require minimal user efforts and automatically supporting new (or updated) applications. Due to thousands of cloud applications use various logic and protocols, cloud access security broker (CASB) solutions (adopted by commercial companies like Skyhigh Networks and CipherCloud [**?**], [7]) struggle to adapt various services by reverse engineering service-specific protocols (further discussed in Sec. II). It is time-consuming and labor intensive. Also, it is difficult to convince cloud providers to reveal their proprietary protocols or participate in collaborations. Therefore, a

practical solution should be able to integrate data protection without reverse engineering or cooperation from cloud providers.

In this paper, we introduce FileCrypt, the first system that automatically and transparently secures sensitive files and preserves user experience and application functionality on browser-based cloud storage, without cooperations from cloud providers or modifying existing applications. With FileCrypt, security conscious users have the choice of uploading encrypted data to cloud applications (e.g., Gmail, Google Drive, Dropbox, Salesforce, etc.) and still use important functionalities (such as search, image preview, and file sharing).

FileCrypt is designed to be able to support different kinds of applications with minimal efforts. Therefore, a critical step is how to automatically capture and identify file operations such as uploading or downloading. With the support of an enterprise proxy, FileCrypt injects JavaScript (JS) snippets to web pages before they are delivered to client browsers. Then the JS snippets overwrite native JavaScript APIs (such as XMLHttpRequest) so that uploading requests can be identified and file content can be encrypted on the fly. This process successfully isolates the sensitive data from the file source and fundamentally address the adaptation issue of various applications. Furthermore, cloud services always perform file integrity checks based on specific parameters (such as file size or file hash) before the cloud servers receive the uploading file, file encryption may cause this step to fail due to inconsistency between encrypted and the original parameters. To address this challenge, FileCrypt replaces the original data with the encrypted data before calculating hash values by overriding the File API of JavaScript.

However, because encrypting data always affects important application functionalities like search operations, we design and integrate a new searchable encryption scheme named *Broker Executed Searchable Encryption* (BESE) into FileCrypt. BESE enables search operations over encrypted data in real applications without any modification on the cloud service. We briefly describe the details of BESE in Sec. III-G.

As a proof of concept, we have implemented FileCrypt and evaluated it with ten real-time browser-based cloud storage applications, including email, storage, office, which demonstrates the effectiveness and generality of our approach. The performance evaluation shows that adopting FileCrypt to protect enterprise sensitive data with fairly low overheads. These applications maintain important functionalities (such as search) while providing transparent encryption.

Our contributions can be summarized as follows:

- We propose FileCrypt, the first system that automatically and transparently protects sensitive files in browser-based cloud storage, which achieves transparent encryption without requiring active cooperation from cloud providers and the modification of existing cloud services. It is scalable and automatically adapts to new applications.
- To ensure essential application functionalities (especially search), while preserving user privacy, we design a new searchable encryption scheme without requiring server-side modification.
- We have implemented FileCrypt and tested it with various popular cloud applications including Gmail, Box, Dropbox, Salesforce, etc. Our experimental results demonstrate that FileCrypt is effective in real-time applications with fairly low overheads.

The remainder of the paper is organized as follows. Section II describes background and related work. Section III discusses the design goals, our threat model, the architecture and the implementation of FileCrypt. Section IV presents performance evaluation and case studies. Section V concludes our work.

II. Background and Related Work

We first present the file uploading models used by browser-based cloud storage services and then discuss related work. We first examine the existing file uploading models by studying their communication protocols.

A. File Uploading Models

- **Initialization Step**. Users need to initiate a file uploading operation and apply for access to the service. Before this operation, a cloud application may verify file names and check available space [8]. It may also check the metadata, which is generated based on the file name, size or hash value on the client side. Please note that not all applications have this step.
- **Uploading Step**. In this step, the application determines whether a file should be sent entirely in one request, or several chunks in multiple requests for performance and scalability. Once the server receives the file, it compares the data against the parameters received in the initialization step; it will cancel the procedure if inconsistency is found. Therefore, if the file is modified during uploading, the operation will be canceled due to the failure of verification.
- **Finalization Step**. Once the uploading procedure is finished, users may have a chance to confirm the upload was successful. This step is usually optional too.

From the above discussion, we can see that encrypting sensitive files and passing file verification in the cloud storage applications is a non-trivial technical challenge.

B. Related Work

A number of existing solutions are proposed to encrypt sensitive data handled by browser-based cloud storage services. In this section, we discuss the pros and cons of each solution and summarize their main differences in Table I.

File Encryption Tools. File encryption applies cryptography to individual files. File encryption tools such as PGP [9] can be used by end users to encrypt and decrypt files locally, which isolates private data from cloud applications. However, users have to manually encrypt files before uploading or decrypt encrypted files after downloading. This solution degrades the user experience and requires users to maintain secret keys for each file.

TABLE I
SOLUTION COMPARISON

Solution	**Encryption Location**	**No client-side deployment**	**Transparent Encryption**	**Nonspecific App Support**	**Key Manager Location**	**Examples**
File Encryption Tools	Client	✗	✗	✓	User	PGP [9], Encryption Dog [10]
Security Overlay	Browser	✗	✓	✗	Key server	Virtru [6], MessageGuard [5]
Cloud Access Security Broker	Proxy	✓	✓	✗	Proxy	Skyhigh [?], CipherCloud [7]
FileCrypt	Browser or proxy	✓	✓	✓	Proxy	

Security Overlay. Security overlay [11] is a technique that provides a window where users can view and interact with secure contents. The solution leverages security overlays to replace the original functionality of an application, which requiring developers to override the functionality. For example, MessageGuard [5] and Virtru [6] create a file upload overlay using HTML iFrames to protect data while sacrificing the mass storage and search functionality. The famous ShadowCrypt [12] replaces input elements in a page with secure, isolated shadow inputs, and encrypted text with secure, isolated cleartext. However, ShadowCrypt only supports encrypting cleartexts and is unable to achieve encryption for files. M-Aegis [13] not only provides isolation but also preserves the user experience through the creation of a conceptual layer called Layer 7.5 (L-7.5), which is interposed between the application (OSI Layer 7) and the user (Layer 8). But M-Aegis only supports encryption for textual data.

Cloud Access Security Brokers. Cloud Access Security Brokers (CASB) [14] focuses on protecting user privacy via a proxy, which adopted by commercial companies like Skyhigh Networks [**?**] and CipherCloud [7]. The proxy sits between a cloud application and a user, where it intercepts and encrypts sensitive data before sending to the cloud. Nevertheless, developers have to specifically integrate with different services through protocol analysis one by one. Also, this approach suffers from the problem that it may not work when the application protocol changes. Indeed, this approach is hard to apply in every cloud applications as it requires a lot of effort to maintain.

Other Work. There are also some proposals investigated encrypting data handled by cloud applications in recent years. Ada Popa proposed CryptDB [15] that transparently encrypts user confidential data between the client side and the database server to adequately protect confidential data. Mylar [16] is based on the Meteor JavaScript framework and builds applications that encrypt all their data sent to the server.

III. FileCrypt Design

A. Design Goals

In this paper, we aim to protect files sent from a trusted computing base to cloud storage via *browser-based interfaces*. As there are already many existing cloud storage services and more will be created, we are interested in finding a solution that is not specific to a given group of services but generic enough to automatically support new ones. In particular, we consider that the intended protection is achieved if the following goals are satisfied:

1. Provide a secure isolated environment where to perform encryption against malicious or compromised cloud providers.
2. Preserve user experience by providing transparent encryption and guarantee rich functionalities including search.
3. Easy to maintain and highly scalable for various applications.

B. Threat Model

The goal of FileCrypt is to ensure the data confidentiality in browser-based cloud storage services. FileCrypt should be deployed in the enterprises network edge whose prime security restricted policies can keep most attackers out. Additionally, The secret kyes stores in the TPM used to prevent malicious insiders who may control the proxy of FileCrypt. Therefore, the internal enterprise network is security and trustworthy. Thus, we assume several parties are not trusted in our threat model:

- Cloud storage service (CSS) providers. CSS providers have strong motivations to compromise user privacy. They may have strong commercial interests or sometimes be required to access sensitive user data by law; they may also be compromised by hackers to steal sensitive data.
- Client-side applications. Client-side application codes from the CSS providers are also considered untrusted.
- Middleware between the CSS and the enterprise premise. The critical files may be compromised outside the corporate network. The middleware between the CSS and the enterprise premise may be exploited to perform man-in-the-middle attacks for exfiltrating sensitive data.

FileCrypt ensures that sensitive file data, which is transmitted over a secure connection or read by client-side code, is encrypted through secure overridden native JavaScript API, even if the client-side code is malicious. Even though the cloud servers conspire, they can not exfiltrate users' private data because they can not get the corresponding key that is located in the Trusted Platform Module (TPM) of FileCrypt. Outside the FileCrypt, even if the user's account is stolen by somebody, the stealer can only access encrypted data, because the encrypted data does not go through the broker and there is no decryption process. We assume the operating systems, browsers, and network devices inside an enterprise firewall are trusted, which is a practical trusted computing base (TCB) in most organizations. FileCrypt does not provide protection against side-channel attacks.

C. FileCrypt Architecture

Fig. 1 illustrates the architecture of FileCrypt. FileCrypt is an internet gateway for on-premises, deployed between enterprise users and cloud service providers to protect outgoing sensitive data. The user still uses common file functions as usual while FileCrypt seamlessly replaces encrypted data read by the application's JavaScript code with clear data

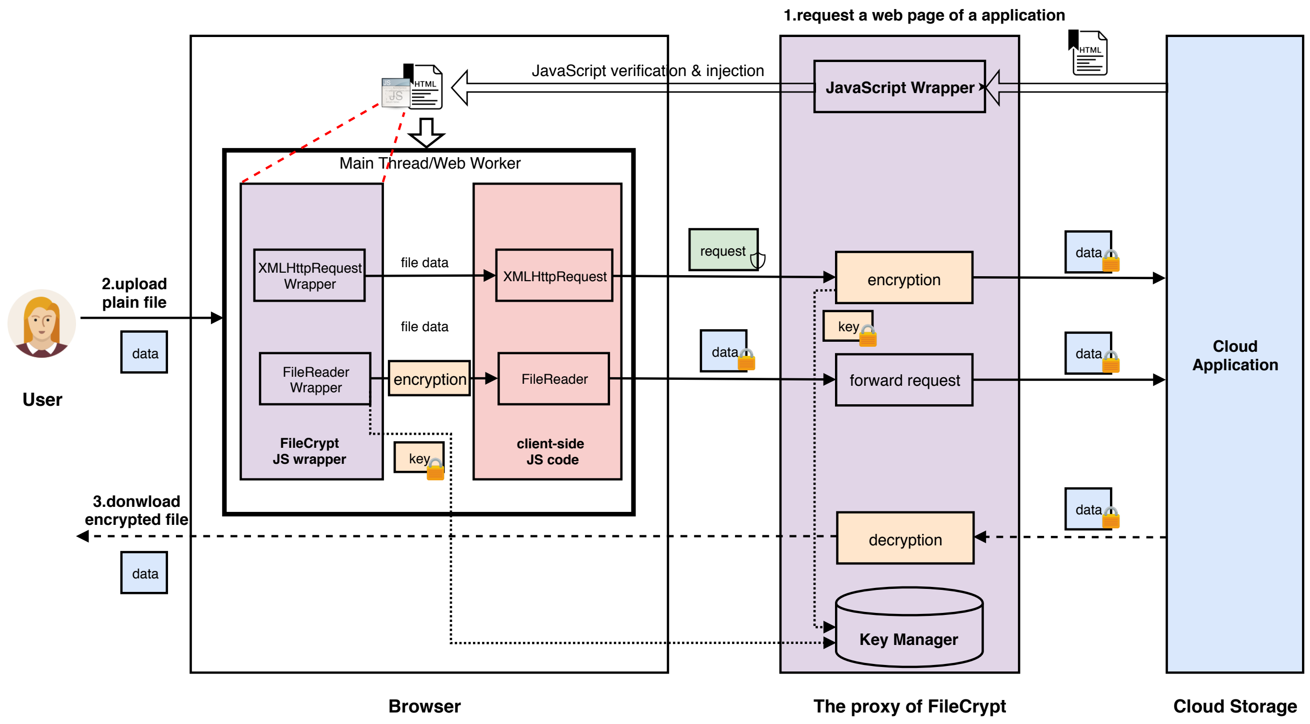

Fig. 1. Architecture of FileCrypt.

appeared in the overwritten APIs. FileCrypt consists of two main components:

1) Secure Proxy: The secure proxy is referred to as the first-mile technology that sits closer to enterprise users. The proxy intercepts HTTP/HTTPS traffic between a browser (also called a client) and the cloud applications via the "SSL man-in-the-middle" technique. FileCrypt acts as a man-in-the-middle component and intercepts the TLS connection with the company trusted certificate. Inspecting traffic is reasonable for enterprises as they must ensure that company data issued by their employees is protected. FileCrypt based on the proxy's ability to inject the JavaScript Wrapper into web pages. Moreover, The proxy is also responsible for automatically identifying encrypted data and decrypting ciphertexts.

2) JavaScript Wrapper: The JavaScript Wrapper is made up of an XMLHttpRequest Wrapper and a FileReader Wrapper. These wrappers are JavaScript code snippets to override the JavaScript native API such as XMLHttpRequest and FileReader. The XMLHttpRequest Wrapper intercepts all the XHR requests then screens and identifies file uploading requests. The FileReader Wrapper encrypts a file by capturing the file read I/O. Although third-party JavaScript libraries can provide various file and web access interfaces, many of them eventually need to invoke the primitive File and Web interface of JavaScript. Meanwhile, the File and XMLHttpRequest API Standard are fairly mature and rarely updated in recent years. So the implementation of FileCrypt can easily keep up with the current standard and has a low risk of out of sync with the existing JavaScript File and Web API. This is a very low-frequency event compared with other development of cloud services. Therefore, we believe this is still a practical solution in the long run.

Let us start with a concrete example. Alice, a user of a cloud storage application browses a web page and the requests go through FileCrypt, which consequently injects the JS Wrapper into the head of the web page. The injected page is executed by the browser, then the JS Wrapper is first executed and installs function hooks to modify the native APIs (such as XHR inside the JavaScript execution environment). It captures calls to JavaScript API functions and object methods in the web page and associated web workers (for a web page to execute scripts in the background).

Algorithm 1 shows the FileCrypt workflow. It first receives an HTTP/HTTPS connection from the client (line 1). The initial request to access the cloud service always includes web pages. It then injects a JS Wrapper into the head of the web pages (line 3). The JS Wrapper will run in the browser and override the native API. The XMLHttpRequest Wrapper starts to intercept network traffic and the FileReader Wrapper captures file operations invoked by the JavaScript code of the application. FileCrypt encrypts the file extracted from the request and rewrites the request body data replaced with the associated ciphertext when the request is labeled with a tag indicating that data need to be protected (line 5). When the signature of the encrypted file returns in the response, the proxy extracts the secret key ID, looks up the right key used to decrypt the ciphertext, then decrypts the ciphertext, rewrites the decrypted data into the response body and returns to the client (line 7).

Algorithm 1 The main FileCrypt algorithm

```
Input: B: the browser
Input: C: the Cloud Storage Service
Input: F: FileCrypt
1: while a HTTP/HTTPS connection c from B do
2:     if c.request contains html page then
3:         c.response ← $inject\_js\_wrapper(c.response)$
4:     if c.request is identified as file uploading then
5:         c.request ← $proxy\_encryption(request)$
6:     if c.response contains the encrypted data then
7:         c.response ← $proxy\_decryption(response)$
8:     send(c)
```

D. Identifing Uploading Requests

One of the most important design goals of FileCrypt is to provide a generic automatic approach for existing and new applications. A fundamental prerequisite for FileCrypt is to adaptively identify the uploading requests of various applications at the proxy. In this section, we first discuss two unsatisfactory approaches and then introduce the dynamic analysis of JavaScript and discuss how we use it in FileCrypt.

Strawman 1. We can simply use a regular expression match to extract features from uploading requests to determine which request satisfies the file uploading request. But this approach requires to accumulate match rules by analyzing all application protocols one by one. Furthermore, these rules in the proxy have to be updated as soon as a cloud service protocol is modified.

Strawman 2. We can also extract an upload request's URL string, HTTP methods, and the corresponding request data using an inter-procedural string analysis, according to [17]. However, service providers usually compress and mix JavaScript codes which lead to low precision of extracting requests in this way.

In fact, the JavaScript codes always invoke the XMLHttpRequest object to send a file in cloud applications. XMLHttpRequest defines a programming interface to transfer data between a web browser and a web server. All modern browsers have a built-in XMLHttpRequest object. The code (shown in Listing 1) defines the file sending procedure with XMLHttpRequest. At line 2, the input element object is saved in variable filesToBeUploaded whose property "files" is a file list. Line 9 passes the file object to the send method of the XMLHttpRequest object. Note that the type of file object in the example can be Blob, File, FormData, and ArrayBuffer. Blob and File always represent the object of immutable file data. FormData consists of a set of key/value pairs representing form fields and their values; they can contain Blob/File Object as values. ArrayBuffer is used to represent a generic, fixed-length raw binary data buffer read from a file. The object type of the file variable (which may be Blob, File or FormData) is enough evidence to determine if an incoming request should be a file uploading request. In the following, we will discuss the conditions when the type of the file is ArrayBuffer.

Therefore, we choose to override the XMLHttpRequest API and add a hook method used to check the argument type in the send method for identifying uploading requests. Listing 2 shows FileCrypt overrides the XMLHttpRequest object with a new XMLHttpRequest object that keeps the original method but can easily intercept and modify XMLHttpRequest requests and responses.

E. Handling File Verification

To deal with data corruption, most cloud storage services validate the uploaded files using either file name, file size, CRC, MD5 hash, and other checksums. If the validation fails, the service replies a failure message. Note that encryption leads to many changes in the file content and length which usually cause validation failures. As mentioned before, an application always submits the initialization parameters like file name, file size, and content checksums to the server before uploading a file. Thus ensuring valid parameters of an encrypted file to pass the validation check is a serious challenge.

The JavaScript File API is used to read the contents of files in the vast majority of modern websites, the JavaScript in a web page will calculate the checksums for the contents and transfer the checksums to the server. Hence, we choose to override the relevant JavaScript API and ensure that uploading encrypted files passes the validation by the cloud server.

FileReader provides efficient ways to access file contents through API functions such as ReadAsArrayBuffer, readAsBinaryString, and readAsText. Here, we use an example to show the handling of readAsArrayBuffer in Listing 3. The reference to the original readAsArrayBuffer is saved in line 1. Line 4 invokes underlying _readAsArrayBuffer method that references to the original readAsArrayBuffer to read the file. In lines 3-8, the 'result' getter in the readAsArrayBuffer is overridden and performs the file encryption. In lines 13-14, the 'resulting' getter of the FileReader is overridden to replace the original data with the encrypted data. When the client-side code in the page invokes the readAsArrayBuffer API to read the file, the application code will obtain the encrypted data returned from the overridden method and calculate the hash values for the encrypted data.

F. Key Management

To enforce the browser-side file encryption transparently and share encrypted files among different enterprises, we introduce a key management mechanism based on Identity-Based Encryption (IBE) [18]. In FileCrypt, each user owns a master key pair (pk_u, sk_u) that is used to encrypt different file keys, where pk_u is the public key and sk_u is the secret key. The encrypted format of the file key is called the wrapper key. In practice, a fully transparent browser-based file encryption framework requires no local storage. Thus users need to delegate secret key storage and management to the trusted intra-enterprise proxy. For security consideration, the proxy

Listing 1. A exmaple of upload file with XMLHttpRequest

```
function uploadFile(){
var filesToBeUploaded = document.getElementById(
    "fileControl");
var file = filesToBeUploaded.files[0];
var xhrObj = new XMLHttpRequest();

xhrObj.open("POST", "upload.cfm", true);
xhrObj.setRequestHeader("Content-type",
    file.type);
xhrObj.setRequestHeader("X_FILE_NAME", file.name
    );
xhrObj.send(file);
}
```

Listing 2. Function hooking via function redefinition

```
var xhr = XMLHttpRequest();
// new a original XMLHttpRequest object
var NativeXMLHttp = XMLHttpRequest;
XHookHttpRequest = function () {
// hooked method and member
}
XMLHttpRequest = XHookHttpRequest;
var newxhr = XMLHttpRequest();
// new a hooked XMLHttpRequest object
```

is supposed to preserve the secret user keys in the Trusted Platform Module (TPM) [19] against adversaries.

Each standalone enterprise holds the public key pairs of different members, we need a mechanism to integrate key information of different parties. Public Key Infrastructure (PKI) is always involved in managing key information in a traditional cryptosystem, but its deployment is a heavy burden for enterprises. To eliminate the overhead of PKI, we leverage advanced IBE schemes to implement a public key cryptosystem. IBE schemes enable Alice to encrypt messages using Bob's identity (e.g., email address), thus there is no need to introduce a third party to manage public keys.

G. Searchable Encryption

While encryption guarantees data security, some functionalities (such as search) are sacrificed in the applications. Searchable encryption schemes achieve different trade-offs between security, usability, and practicability. More academic projects focus on index-based Searchable Encryption (SE), in which the cloud can search the encrypted index with search trapdoor generated by the user and return encrypted documents [20]–[25]. We name this approach as Cloud Executed Searchable Encryption (CESE). But CESE requires modifications on the cloud, which is difficult in practice. We present a new searchable encryption scheme called *Broker Executed Searchable Encryption (BESE)*. The BESE scheme builds the index in the broker (proxy) with identifiers pointing to encrypted data in the cloud servers. After FileCrypt uploads the encrypted file to the cloud, the cloud will return the file identifier. With buffered data before encryption and the encrypted file identifier, FileCrypt can index the data, associate it with an encrypted file identifier in the proxy and storage it on a remote

Listing 3. Function hooking via function redefinition

```
FileReader.prototype._readAsArrayBuffer =
    FileReader.prototype.readAsArrayBuffer;

FileReader.prototype.readAsArrayBuffer =
    function readAsArrayBuffer () {
  this._readAsArrayBuffer.apply(this, arguments)
    ;
  Object.defineProperty(FileReader.prototype, '
    result', {
    get: function () {
      var string = this.resultString;
      // encrypting data
      var result = encryption(string);
      return result;
    }
  }
}

Object.defineProperty(FileReader.prototype, '
    resultString', {
  get: FileReader.prototype.__lookupGetter__('
    result')
});
```

cloud storage server as shown in Algorithm 2. The details of BESE is depicted as follows:

1) A user uploads a document D to the cloud storage application C_1.
2) FileCrypt intercepts and encrypts the document D with the file key K, then gets the uploaded file identifier $ID(D')$ returned from C_1.
3) Next, FileCrypt extracts keywords ($\{w_1, w_2, ..., w_n\}$) of the document D cached in the proxy. It integrates the searchable tokens of keywords into a index file $I = \{t_1, t_2, ..., t_n\}$ and uploads the encrypted index I to the the remote cloud storage server C_2 via cloud storage API.
4) FileCrypt maps the index I with corresponding index file identifier $ID(I)$ and associates $ID(I)$ with corresponding encrypted file identifier $ID(D')$.
5) When a user performs a search for a keyword w to initiate a query request, the keyword w is replaced with searchable token TK_w and searched on C_2.
6) The service C_2 returns the index file identifier id. Then FileCrypt obtains the encrypted file identifiers id' from the mappings.
7) The identifier id' can be used to request the encrypted files D' from the storage application C_1.

H. FileCrypt Implementation

We have implemented FileCrypt as an enterprise gateway prototype, which has been initially tested by several companies. We will consider making our software available to the public in the future. FileCrypt is implemented based on Squid [26], which is a popular open-source caching proxy for the web supporting HTTP, HTTPS, and FTP. The JavaScript Wrapper of FileCrypt can be easily implemented by overriding XMLHttpRequest and FileReader API. We believe the JavaScript Wrapper should work well on any other browsers since we adopt standard native API in JavaScript.

Algorithm 2 BESE algorithm

```
$D = \{w_1, w_2, ..., w_n\}$ A document contains $n$ words
$D'$ The encrypted file of $D$
$k$ The secret key k used to generate searchable tokens
$K$ The file key
$K'$ The wrapper key
$E_k(D)$ A symmetric encryption scheme
$ID(D)$ The identifier of document $D$
$f_k(w_i)$ A pseudo-random function
$I$ The encrypted index

$\boldsymbol{KeyGen(1^s)}$
    Given a security parameter s, output keys $K, k$.
$\boldsymbol{BuildIndex(K, D)}$
1: Compute $D' = E_k(D)$ and upload it to the cloud $C_1$;
2: Compute $K' = E_{pk_u}(K)$ and store it in the proxy;
3: Get the document identifier $ID(D')$ returned from $C_1$;
4: for each keyword $w_i$ in $D$ do
5:     token $t_i = f_k(w_i)$
6: Build the index file $I = \{t_1, t_2, ..., t_n\}$ and upload $I$ to the server $C_2$;
7: Get the index identifier $ID(I)$ returned from $C_2$ and build the mapping $Map_1 < ID(I), I >$ and $Map_2 < ID(I), ID(D') >$
8: Output the index and the mapping $(I, Map_1, Map_2)$.
$\boldsymbol{Trapdoor(w)}$
1: Output $TK_w = f_k(w)$.
$\boldsymbol{Search(I, TK_w)}$
1: Search $TK_w$ in the index $I$ of $C_2$ and get the corresponding index identifiers $\{ID(I_1), ID(I_2)...\}$.
2: Output the corresponding document identifiers $\{ID(D'_1), ID(D'_2)...\}$ from the mappings according to the above index identifiers.
```

FileCrypt encrypts files using the AES algorithm in the CTR mode. Since the AES CTR mode outperforms the CBC mode and does not require the plaintext to be encrypted in blocks, it guarantees that the length of the plaintext is always the same as the length of the ciphertext. We use the Stanford JavaScript Crypto Library (SJCL) [27] to perform encryption in the JavaScript Wrapper and adopt the OpenSSL library to perform encryption in the proxy. The filename of the encrypted file like "YAB-098...7F6", includes a formatted signature and the ID of the wrapped key. 'YAB' is the formatted signature, which explicitly marks encrypted files. The wrapper key is an encoded format of file key encrypted by the user's secret key. FileCrypt adopts Stanford IBE library [28] to protect file keys discussed in Sec. III-F. We also utilized HMAC-SHA-256 to generate searchable tokens in BESE.

IV. Experimental Evaluation

In this section, we measure the performance overhead of FileCrypt and discuss its effectiveness in a wide variety of popular applications. The experimental setup consists of a client machine that has Intel i7 2.20GHz CPU with 4 cores and 16 GB RAM, and a virtual machine with Intel i7 2.20GHz CPU with 2 cores and 4 GB RAM, which running FileCrypt. We simulate users interactions with an application via FileCrypt on the client machine.

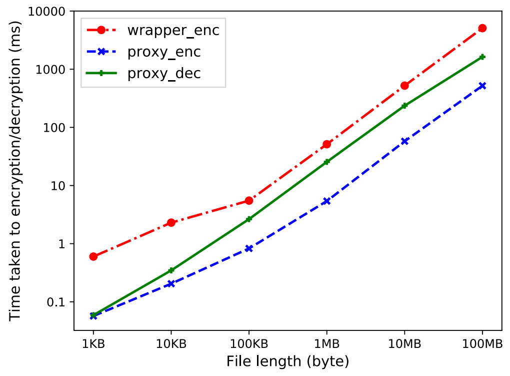

Fig. 2. Time taken for FileCrypt to encrypt/decrypt a file. "wrapper_enc" is the encryption time with the JavaScript Wrapper. "proxy_enc" is the encryption time of the proxy. "proxy_enc" is the decryption time of the proxy.

A. Performance Evaluation

We first measure the overhead introduced by cryptographic operations from FileCrypt. The JavaScript Wrapper performs encryption before the data read by the client-side code. The proxy encrypts data before sending to the application. We invoked the FileCrypt cryptographic interfaces to encrypt plaintexts and decrypt ciphertexts, then measured the delays as we varied the message length from 1 KB to 100 MB. Fig. 2 shows the median time overhead for 100 trials for each case. The JavaScript Wrapper shows an overhead of 5089.3 ms to encrypt a 100MB file. However, most applications adapt to upload files in smaller file chunks with a maximum size that less than 10 MB. A file reading with a chunk size of 100 KB only takes an extra 5.5 ms, which is not noticeable for users. The proxy consumes 520.311 ms to a 100-MB file and 1631.370 ms to decrypt a 100-MB file. Clearly, the overhead introduced by cryptographic operations is relatively small compared with the total delay of file uploading or downloading.

Next, we test FileCrypt with five major applications (for a wide range of users) to evaluate its overheads. Among above applications, Gmail and QQMail are the popular email services which can transfer email attachments; Box and Dropbox provide lots of users and businesses with the ability to store, access easily, and share files and folders; Google Docs and Salesforce are most popular services with cloud storage among companies. Because the use of a proxy server is very common in enterprise networks to protect the internal machines, we just compare the overhead of the proxy that just forwards traffic with the overhead of FileCrypt which encrypts files. To estimate FileCrypt overhead for real-world applications, we upload files that vary from 1 KB to 100 MB to the service and download the encrypted file from the service with the

TABLE II
THE PERFORMANCE OF FILECRYPT

Cloud App	File upload interface	Verification	File size	Uploading			Downloading		
				w/ Proxy (ms)	w/ FileCrypt (ms)	Overhead	w/ Proxy (ms)	w/ FileCrypt (ms)	Overhead
MailQQ	send(ArrayBuffer)	size,checksum	1KB	187.89	190.41	1.30%	34.3	25.5	10.52%
			10KB	256.50	262.20	2.17%	45	49.2	8.53%
			100KB	397.09	408.26	2.73%	129.6	138.2	0.62%
			1MB	1381.80	1417.90	2.55%	1145.7	1168.5	1.95%
			10MB	8654.71	9612.43	9.96%	1480.20	1496	1.05%
			100MB	106321.70	118351.82	10.16%	10319.40	11368.60	9.23%
Box	send(File)	size	1KB	1366	1532	10.83%	792.2	799.4	0.90%
			10KB	2291	2559	10.47%	2318.40	2460.80	5.78%
			100KB	3907	3954	1.19%	2701.4	2825.3	4.38%
			1MB	3924	4132	5.03%	4878.80	5300.20	7.95%
			10MB	12578	12919	2.64%	6332.80	6561.60	3.48%
			100MB	235714.30	249600	5.56%	22209.60	23120.80	3.94%
Dropbox	send(File)	size,checksum	1KB	2069	2261	8.49%	384.8	390.2	1.38%
			10KB	1595	1667	4.31%	517.20	528	2.04%
			100KB	1592	1627	2.15%	1638	1771.2	7.52%
			1MB	1410.77	1466.44	3.79%	3113.20	3181.80	2.15%
			10MB	7223	77891	7.26%	11270.80	11679.40	3.50%
			100MB	295116	331158.30	10.88%	23403.60	24701.40	5.25%
Google Docs	send(Blob)	size	1KB	644.8	714.1	9.70%	1410.20	1422	0.829%
			10KB	1143	1205	5.14%	1748	1830.40	4.50%
			100KB	1908	2039	6.42%	2388.60	2411.40	0.945%
			1MB	4771	4877	2.17%	2768.80	2800.60	1.13%
			10MB	11189	12043	7.09%	3659.40	3865	5.32%
			100MB	87420	97800	10.61%	18140.2	18575.20	2.34%
Salesforce	send(FormData)	size	1KB	575.89	577.07	8.87%	1410.20	1422	0.83%
			10KB	578.19	587.09	1.52%	725.4	744.5	2.56%
			100KB	986.35	990.61	0.43%	985.3	990.7	0.54%
			1MB	1961	1999	1.90%	1815.5	1919	5.39%
			10MB	9823	10117	2.91%	25231	26364	4.30%
			100MB	142800	153000	6.67%	176071	183848	4.23%

proxy or with the FileCrypt 10 times, then take the average delay. As shown in the Table. II, these operations usually are in milliseconds. Comparing the delay for the same operation in the FileCrypt with the proxy, we observe that FileCrypt has low overheads and could be applied in real-world applications. Cryptographic operations introduce negligible overheads (less than 11%) for a file operation in the cloud. We find that there is a downloading overhead of 10.52% for 1KB and 9.23% for 100MB, but all the intermediate values have a much lower overhead. This significant difference is due to that network environment changes in real time, which have much more impact on small files. In addition, as the data size increases, a large number of memory operations occur in the code, causing performance degradation. So we believe that continuous code optimization will considerably improve the performance. Besides, the performance of search is 11 operations per second, indicating that the latency of searching is negligible.

B. Case Study

We test the FileCrypt on a wide variety of applications that handle files based on browsers, which is the focus of FileCrypt. We discuss these applications that retained or lost functionality when using them with FileCrypt. The result shows that FileCrypt retains some important functionality of these applications while protecting sensitive critical data.

MailQQ [29] is an email service developed by Tencent. The service provides input fields including the subject field, the body field, and email attachments. The implementation of MailQQ's file uploading is the most complicated in the applications we have tested. Moreover, the MailQQ checks the file size and the chunk size strictly and it is hard to extract and encrypt files by direct protocol and code analysis. The JS Wrapper injected by FileCrypt encrypts a file before loading by the client-side code from the application, which ensures that the MD5 checksum computed with the uploaded data is the encrypted file's MD5. Nonetheless, this demonstrated that FileCrypt is scalable to applications with a complicated implementation on file uploading with no additional effort. Since the server does not parse the encrypted content of a file, users cannot preview and edit documents like PDF and WORD stored in the cloud. Although losing document preview and editing functions, it deserves to prevent cloud providers from peeping and leaking the users' data. We also used **Gmail** [30] with FileCrypt, and found that encrypting email attachments did break the Gmail's document editing feature. The future of document editing and previewing are provided with cloud applications by parsing the original documents and compressing the original images. The user needs to pay a little price for keeping sensitive data safe from prying eyes of compromised cloud providers.

We also tested FileCrypt with typical browser-based cloud storage applications such as **Dropbox** [31], **Box** [32],

OneDrive [33], **Google Drive** [34], and **Mega.nz** [35]. FileCrypt successfully encrypts uploaded files and decrypts encrypted files downloaded from the cloud. Two broken features are the document preview and sharing files with the outside of an enterprise network. In some cases, companies want to encrypt the sensitive data before it leaves their firewalls, so they prefer FileCrypt to sit on premises and handle encryption, decryption, and sharing of data within their networks.

On using FileCrypt with office applications, namely **Salesforce** [36], **Google Docs** [37], and **Slack** [38]. The uploaded files of Salesforce and Google Docs can be encrypted by FileCrypt. Importing a file into a trading repository and report forms did not work with Salesforce due to the encrypted imported data. Fortunately, users can choose whether to enable encryption by clicking the hovering button in the head of the web page. Further, we found that Slack relies on WebSocket connections to transfer files, and we believe that our solution can extend to data encryption based on WebSocket in future work.

V. Conclusion

In this paper, we present FileCrypt, a generic approach that automatically performs file encryption on various cloud applications while preserving most file storage functionality. As a result, it has almost no impact on user experiences and can help users against data breaches by malicious or compromised cloud service providers and attackers. Our experimental results show that FileCrypt has relatively low overheads and can support practical use in many real-world applications.

VI. Acknowledgment

This research is supported by the National Key Research and Development Program of China (No. 2016YFB0800803) and the National Natural Science Foundation of China (No. 61872110).

References

[1] "iCloud leaks of celebrity photos." [Online]. Available: https://en.wikipedia.org/wiki/ICloud_leaks_of_celebrity_photos

[2] "Your Sensitive Information Could Be at Risk: File Sync and Share Security Issue," Publishing Date: May 6, 2014. [Online]. Available: https://blogs.intralinks.com/2014/05/sensitive-information-risk-file-sync-share-security-issue/

[3] "Doxed by Microsofts Docs.com: Users unwittingly shared sensitive docs publicly," Publishing Date: March 27, 2017. [Online]. Available: https://arstechnica.com/security/2017/03/doxed-by-microsofts-docs-com-users-unwittingly-shared-sensitive-docs-publicly/

[4] "Skyhigh: 9 cloud computing security risks every company faces." [Online]. Available: https://www.skyhighnetworks.com/cloud-security-blog/9-cloud-computing-security-risks-every-company-faces/

[5] S. Ruoti, J. Andersen, T. Monson, D. Zappala, and K. E. Seamons, "Messageguard: A browser-based platform for usable, content-based encryption research," *CoRR*, vol. abs/1510.08943, 2015. [Online]. Available: http://arxiv.org/abs/1510.08943

[6] "Virtru: Email encryption and data security for business privacy." [Online]. Available: https://www.virtru.com

[7] "Ciphercloud: cloud services adoption while ensuring security, compliance and control." [Online]. Available: https://ciphercloud.com/

[8] H. Chen, L.-J. Zhang, B. Hu, S.-Z. Long, and L.-H. Luo, "On developing and deploying large-file upload services of personal cloud storage," in *Services Computing (SCC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 371–378.

[9] "Symantec desktop email encryption end-to-end email encryption software for laptops and desktops." [Online]. Available: http://www.symantec.com/desktop-email-encryption

[10] "Folder encryption dog: Encrypt your folder and maintain the privacy of your confidential data." [Online]. Available: http://soarersoft.com/dirwatchdog.htm

[11] S. Ruoti, N. Kim, B. Burgon, T. Van Der Horst, and K. Seamons, "Confused johnny: when automatic encryption leads to confusion and mistakes," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*. ACM, 2013, p. 5.

[12] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "Shadowcrypt: Encrypted web applications for everyone," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1028–1039. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660326

[13] B. Lau, S. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva, "Mimesis aegis: A mimicry privacy shield–a systems approach to data privacy on public cloud," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 33–48.

[14] "Cloud access security brokers." [Online]. Available: https://www.gartner.com/it-glossary/cloud-access-security-brokers-casbs/

[15] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.

[16] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building web applications on top of encrypted data using mylar." in *NSDI*, 2014, pp. 157–172.

[17] E. Wittern, A. T. Ying, Y. Zheng, J. Dolby, and J. A. Laredo, "Statically checking web api requests in javascript," in *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 244–254.

[18] A. Shamir, *Identity-Based Cryptosystems and Signature Schemes*. Springer Berlin Heidelberg, 1984.

[19] T. C. Group, "Tpm," http://www.trustedcomputinggroup.org.

[20] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *International conference on the theory and applications of cryptographic techniques*. Springer, 2004, pp. 506–522.

[21] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 965–976.

[22] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[23] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 442–455.

[24] E.-J. Goh *et al.*, "Secure indexes." *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[25] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000, pp. 44–55.

[26] "Squid." [Online]. Available: http://www.squid-cache.org/

[27] Stanford, "Stanford javascript crypto library," https://crypto.stanford.edu/sjcl/.

[28] "Stanford ibe library." [Online]. Available: https://crypto.stanford.edu/ibe/

[29] "Mailqq." [Online]. Available: https://mail.qq.com/

[30] "Gmail." [Online]. Available: https://mail.google.com

[31] "Dropbox: Cloud storage service." [Online]. Available: https://www.dropbox.com/

[32] "Box: Cloud storage service." [Online]. Available: https://www.box.com/

[33] "Onedrive." [Online]. Available: https://onedrive.live.com/

[34] "Googledrive." [Online]. Available: https://drive.google.com

[35] "Mega.nz." [Online]. Available: https://mega.nz/

[36] "Salesforce." [Online]. Available: https://www.salesforce.com

[37] "Googledocs." [Online]. Available: https://docs.google.com

[38] "Slack." [Online]. Available: https://slack.com/