SafeSky: A Secure Cloud Storage Middleware for End-user Applications

Rui Zhao[1], Chuan Yue[1], Byungchul Tak[2], and Chunqiang Tang[3]

[1]Colorado School of Mines
[2]IBM Research Division Thomas J. Watson Research Center
[3]Facebook Inc.

***Abstract*—As the popularity of cloud storage services grows rapidly, it is desirable and even essential for both legacy and new end-user applications to have the cloud storage capability to improve their functionality, usability, and accessibility. However, incorporating the cloud storage capability into applications must be done in a secure manner to ensure the confidentiality, integrity, and availability of users' data in the cloud. Unfortunately, it is non-trivial for ordinary application developers to either enhance legacy applications or build new applications to properly have the secure cloud storage capability, due to the development efforts involved as well as the security knowledge and skills required. In this paper, we propose SafeSky, a middleware that can immediately enable an application to use the cloud storage services securely and efficiently, without any code modification or recompilation. A SafeSky-enabled application does not need to save a user's data to the local disk, but instead securely saves them to different cloud storage services to significantly enhance the data security. We have implemented SafeSky as a shared library on Linux. SafeSky supports applications written in different languages, supports various popular cloud storage services, and supports common user authentication methods used by those services. Our evaluation and analysis of SafeSky with real-world applications demonstrate that SafeSky is a feasible and practical approach for equipping end-user applications with the secure cloud storage capability.**

***Keywords*—Applications; Cloud storage; Middleware; Security;**

I. Introduction

Cloud computing is a significant trend and it can offer many benefits such as cost efficiency, elasticity, scalability, and convenience to millions of organizations and end users. For many applications, especially end-user applications, it is often desirable and even essential to have the *cloud storage capability* to enhance their functionality, usability, and accessibility. For example, document processing applications may want to save users' sensitive documents to the cloud, accounting or healthcare applications may want to save users' financial or health information to the cloud, and Web browsers may want to save users' browsing data such as bookmarks and histories to the cloud [23]; in all these cases, one considerable benefit to users is that their data stored in the cloud can be available and readily usable anytime, anyplace, and on any computer.

However, one of the major concerns that inhibits the cloud adoption is security [2], [3], [18]. Not only many new security problems such as unexpected side channels and covert channels as well as insider attacks can occur in the cloud, but also organizations and end users do not have sufficient confidence in hosting sensitive data in the cloud. Therefore, the *cloud storage capability* must be securely equipped to end-user applications (referred to as having the ***secure cloud storage capability***) to ensure the confidentiality, integrity, and availability of the data saved to the cloud.

Unfortunately, it is nontrivial for ordinary application developers to either enhance legacy applications or build new applications to properly have the secure cloud storage capability. The complexity of both applications and cloud storage services often requires deep domain expertise from developers, thus mandating a substantial development effort for the cloud storage capability integration. Moreover, a lack of sufficient security knowledge and skills in application developers can often incur design, implementation, and deployment vulnerabilities as shown in many studies [9], [12], [24].

Researchers have proposed many systems to continuously improve the security, reliability, and availability of cloud storage services [1], [7], [8], [13], [14], [16], [21]; however, merely focusing on the sever-end enhancement is insufficient because a particular cloud storage service may still be compromised by outsider or insider attackers. Therefore, to provide a strong security guarantee, applications must properly protect users' data at the user-end in the first place. Like us, some researchers have realized the importance of facilitating end-user applications to have the secure cloud storage capability [5], [20]; however, those solutions suffer from the deployment and usage limitations as discussed in Section II.

In this paper, we take a middleware approach and design SafeSky, a secure cloud storage middleware that can immediately enable either legacy or new end-user applications to have the secure cloud storage capability without requiring any code modification or recompilation to them. SafeSky is designed as a middleware library that can be dynamically loaded with different applications; it sits between the applications and the operating system to intercept the applications' file operations and transform them into secure cloud storage operations. To integrate this middleware into an application, developers or even advanced users can simply copy the SafeSky library and create a corresponding command for starting the application with the library. A SafeSky-enabled application does not need to save any data to the local disk, but instead securely saves the data to multiple free cloud storage services to simultaneously enhance the data confidentiality, integrity, and availability. To use a SafeSky-enabled application, end users simply need to provide their cloud storage accounts to SafeSky at the beginning of each application session, while SafeSky will transparently take care of everything else behind the scenes.

We have implemented SafeSky in C and built it into a shared library on Linux. It supports applications written in languages such as C, Java, and Python as long as they interact with the underlying operating system through the dynamically linked GNU *libc* library. It supports popular cloud storage services such as Amazon Cloud Drive, Box, Dropbox, Google Drive, Microsoft OneDrive, and Rackspace; it also supports common user authentication methods used by the popular cloud storage services. We have evaluated the correctness and performance of SafeSky by using three real-world applications: HomeBank, SciTE Text Editor, and Firefox Web browser; we have also analyzed the security of SafeSky. Our evaluation and analysis results demonstrate that SafeSky is a feasible and practical approach for equipping end-user applications with the secure cloud storage capability.

The main contributions of this work include: (1) a novel middleware approach for immediately enabling either legacy or new end-user applications to have the secure cloud storage capability without requiring any code modification or recompilation (Section III); (2) a concrete SafeSky middleware system for flexibly supporting diverse end-user applications, cloud storage services, and authentication methods (Sections III and IV); (3) an evaluation of SafeSky using real-world applications (Section V); (4) a security analysis of SafeSky (Section VI).

II. Background and Related Work

A large number of cloud storage services have been deployed and widely used [25], [26], [27], [28], [30], [31], [33]. Most cloud storage services offer free accounts and storage spaces to regular users, and many of them follow the predominant REST (Representational State Transfer) Web service design model [11], [15] and allow different client applications to easily access them through their REST APIs. Organizations and advanced users can also deploy their own cloud storage services. For example, one popular cloud storage software is OpenStack Swift [37], which is free and also provides REST APIs to client applications. Note that we do not intend to build any new cloud storage service, but focus on enabling SafeSky to directly use these widely deployed and easily accessible cloud storage services.

As highlighted in Section I, having the cloud storage capability is desirable and even essential for many end-user applications to provide better functionality, usability, and accessibility to users. Existing end-user applications (e.g., for document processing, accounting, healthcare, task scheduling, contact management, and browsing) as well as the potential future applications can all use the cloud storage capability to benefit users by enabling them to conveniently access their data anytime, anyplace, and on any computer. However, this considerable benefit does not come without the risks of losing data confidentiality, integrity, and availability. The recent leak of celebrity photos in iCloud [32] is just one of the numerous reported or even unreported data breaches.

Vendors and researchers have proposed a number of systems to continuously improve the reliability, availability, and security of cloud storage services. Popa et al. proposed CloudProof, a secure storage system that enables customers to detect violations of data integrity, write-serializability, and freshness in the cloud [16]. Wang et al. proposed a distributed storage verification scheme to ensure the correctness and availability of cloud data [21]. Kamara and Lauter proposed a virtual private storage service to combine the security benefits of using private clouds with the availability and reliability benefits of using public clouds [13]. Mahajan et al. proposed Depot, a cloud storage system that provides safety and liveness guarantees to clients without even requiring them to trust the correctness of Depot servers [14]. The Windows Azure team developed a highly available cloud storage architecture as described in [8].

Researchers have also emphasized the importance of incorporating redundancy into the cloud storage services to further improve their reliability, availability, and security. Bowers et al. proposed HAIL, a distributed cryptographic system that applies RAID (Redundant Arrays of Inexpensive Disks)-like techniques to achieve high-availability and integrity across cloud storage providers, and allows servers to prove to a client that a stored file is intact and retrievable [7]. Abu-Libdeh et al. proposed RACS, a proxy that also applies RAID-like techniques, but focuses on transparently using multiple providers to achieve cloud storage diversity, avoid vendor lock-in, and better tolerate provider outages or failures [1].

However, merely focusing on the sever-end enhancements is insufficient because a particular cloud storage service may still be compromised by outsider or insider attackers [32]. In addition, end users should also consider the risks of cloud service vendor lock-in [1], [3]. Therefore, to provide a strong security guarantee, applications must properly protect users' data at the user-end in the first place. Like us, some researchers have realized the importance of facilitating end-user applications to have the secure cloud storage capability. They have explored the API library approach [5] and the file system proxy approach [20] reviewed as below.

Bessani et al. proposed DepSky, a system that sits on top of multiple cloud storage services to form a cloud-of-clouds [5] and applies the Shamir's (k, n) secret sharing scheme [17] to improve the overall data availability and confidentiality. We also emphasize the importance of incorporating redundancy, and DepSky is more similar to our SafeSky in terms of applying the Shamir's (k, n) secret sharing scheme to achieve a high-level security and availability. However, DepSky took an API library approach and requires developers to use its APIs to modify their code; therefore, it still suffers from the problem that developers may misuse APIs and may fail to follow secure design, implementation, and deployment practices [9], [12], [24]. In contrast, our SafeSky can enable either legacy or new end-user applications to immediately have the secure cloud storage capability without requiring any code modification or recompilation to them, thus bringing important deployment and security benefits.

Another work, BlueSky [20], is similar to our SafeSky in terms of not requiring any application modification. However, BlueSky is a file system proxy that aims to lower the cost and improve the performance of using cloud storage services by adopting a log-structured data layout for the file system stored in the cloud [20]. Thus, its design requirements and decisions are different from those of SafeSky that put security as the first priority. Furthermore, its file system proxy approach is heavier than our middleware approach because clients need to mount the BlueSky file systems, which need to be properly set up and maintained by system administrators; therefore, it is more appropriate for using BlueSky to provide services to clients in enterprise environments [20]. Our SafeSky is informed by traditional cryptographic file systems such as [6], [22], but it is a lightweight cloud-oriented middleware that can be simply incorporated by developers and individual end users into their applications.

III. Design

Our objective is to design a secure cloud storage middleware, SafeSky, that can immediately enable either legacy or new end-user applications to have the secure cloud storage capability without requiring any code modification or recompilation to them. A SafeSky-enabled application does not need to save any data to the local disk, but instead securely saves the data to multiple free cloud storage services to simultaneously enhance the data confidentiality, integrity, and availability.

A. Threat Model and Assumptions

The basic threat model that we consider in the design of SafeSky is that attackers can obtain users' data saved in a particular cloud storage service and may then further compromise the data confidentiality, integrity, and availability. Attackers could be outsider unauthorized or illegitimate entities who initiate attacks from outside of the security perimeter of a cloud storage service; examples of ***outsider attackers*** could be from amateur pranksters to organized criminals and even hostile governments. Attackers could also be insider entities who are authorized to access certain resources of a cloud storage service, but use them in a non-approved way; examples of ***insider attackers*** could be insincere or former employees who can still access the resources of a cloud storage service. We do not aim to prevent the stealing of users' data saved in a cloud storage service, a goal that is difficult to achieve given the many data breaches reported everyday. Instead, we focus on ensuring that it is computationally or even absolutely infeasible for attackers to decrypt and use the data stolen from a particular cloud storage service.

We assume that on a user's computer, the operating system is secure and no malware is installed to steal the user's data, for example, from memory or input devices; meanwhile, SafeSky itself is not compromised because it is part of the trusted computing base of the system. We assume that in the cloud, multiple storage service providers do not collude to compromise the security of a user's data; meanwhile, a user's multiple cloud accounts are not compromised at the same time (e.g., due to shared or weak passwords) by attackers for them to further steal the user's data. In addition, if an application directly transmits a user's data to a server through network connections, SafeSky does not protect the security of such data because manipulating network transmissions can easily break the functionality and semantics of the application.

B. Requirements and Challenges

To achieve our objective, we identify five key design requirements for SafeSky: **(1)** ***confidentiality and integrity***: Users' data often contain highly sensitive information, and may determine the execution logic of applications. Therefore, SafeSky must securely protect the data at the user-end before saving them to cloud storage services, so that it is computationally or even absolutely infeasible for either outsider or insider attackers to compromise the data confidentiality and integrity. **(2)** ***availability***: Saving the data to the cloud can benefit users for accessing the data from different places and computers, but it may suffer from the problem that some cloud storage services could be unavailable occasionally. Therefore, SafeSky needs to ensure high data availability, so that applications can access their data anytime even if certain cloud storage services are unavailable. **(3)** ***deployability***: Incorporating the secure cloud storage capability into applications could be a challenging task for many developers and could be error-prone. Therefore, SafeSky must be easily deployable, so that different applications can immediately have the secure cloud storage capability without requiring any code modification or recompilation to them. **(4)** ***consistency***: SafeSky should satisfy the single-reader single-writer consistency semantics for supporting single-user applications that are most widely used. **(5)** ***performance***: SafeSky should not incur any perceivable performance overhead to end users.

These requirements bring a few ***challenges*** to the design and implementation of SafeSky. Simultaneously achieving the three security requirements confidentiality, integrity, and availability in user data protection is the foremost challenge because we need to properly choose and synthesize different cryptographic primitives, and consider both insider and outsider attackers. Making SafeSky easily deployable and transparent to applications is the second major challenge because we need to consider a variety of file operations that could be issued by different applications. Ensuring the consistency and efficient access of users' data in the cloud is the third major challenge because we need to consider the heterogeneous nature of cloud storage services in terms of their different user authentication methods and application programming interfaces, and the diverse workload characteristics of different applications.

C. Overview and Rationale

Figure 1 illustrates the high-level architecture of SafeSky. It consists of three components: *interception*, *data protection*, and *cloud driver*. Originally without SafeSky (as shown in the left dashed box in Figure 1), to perform local disk file operations, applications invoke function or system calls through C libraries such as the GNU *libc* library. Note that any Unix-like or Linux-like operating system needs a C library. With SafeSky, applications perform local disk file operations as usual, while SafeSky intercepts the original file operations in its interception component, protects the intercepted data in its data protection component, and saves the protected data to multiple cloud storage services in its cloud driver component. The applications can be implemented in languages such as C, Java, and Python as long as they interact with the underlying operating system through the dynamically linked C libraries.

We design SafeSky as a middleware library that can be dynamically loaded with different applications. To integrate this middleware into an application, developers or advanced users simply need to copy the SafeSky library and create a corresponding command for starting the application with the SafeSky library dynamically loaded before other libraries. To use the SafeSky-enabled application, end users simply need to provide their cloud storage accounts to SafeSky at the beginning of each application session, while SafeSky will transparently take care of everything else behind the scenes.

The interception component intercepts applications' text and binary file operations either at the standard C function

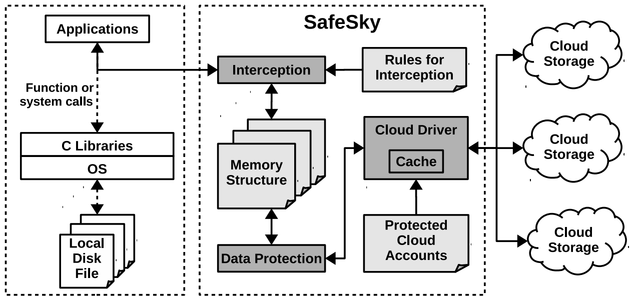


Fig. 1: High-level architecture of SafeSky

level (e.g. the buffered fread() and fwrite() functions) or at the system call wrapper function level (e.g. the unbuffered read() and write() functions), and manages the intercepted data with block-level granularity for each file in a memory structure. Interception at either of those two levels has its own advantages and disadvantages. Interception at the standard C function level has the platform independence benefit and can immediately support the applications to run on different operating systems, but it does not support the applications that do not use the standard C functions. Interception at the system call wrapper function level can immediately support different applications (on a given operating system) regardless of whether they use standard C functions or not, but it does not support the applications that run on other operating systems. If an application calls standard C functions, SafeSky performs the interception at the standard C function level; otherwise, it performs the interception at the system call wrapper function level. Therefore, different applications on different platforms can be flexibly supported by SafeSky, no matter they use one or both types of functions.

The data protection component securely protects each new or updated data block in the memory structures before letting the cloud driver component send the data block to the cloud. It first applies the authenticated encryption to a new or updated data block in a memory structure (for a file) to ensure that it is computationally infeasible for attackers to break the data confidentiality and integrity. Furthermore, it applies the Shamir's (k, n) secret sharing scheme [17] to split the protected data block as well as the corresponding authenticated encryption key and parameters into n pieces for saving to n different cloud storage services, so that it is ***absolutely infeasible*** [17] for attackers to break the data confidentiality, given that they do not compromise k or more cloud storage services at the same time. Using this secret sharing scheme also ensures high data availability because the protected data blocks and the keys can be reconstructed from any k available cloud storage services.

The cloud driver component saves/retrieves the split data block pieces to/from different cloud storage services. At the beginning of each application session, this component authenticates a user to the cloud storage services using the user's protected cloud accounts. Within the session, when a data block is read for the first time by the application, this component retrieves k data block pieces from any k of the n cloud storage services to reconstruct the protected data block, which will be authenticated and decrypted by the data protection component to recover that data block; this retrieval operation will only occur once in a session for each data block of each file. Whenever a data block of a file is created or updated by the application and its n data block pieces are generated by the data protection component, the cloud driver component saves those data block pieces to n different cloud storage services. We refer to this "one retrieval and multiple saves" mechanism as ***saves-after-retrieval***.

This high-level architecture is a rational design for SafeSky to meet those five key requirements and address those major design and implementation challenges (Section III-B). It applies authenticated encryption and secret sharing schemes to ensure the data security in the cloud. It uses dynamic loading techniques and supports the file operation interception at two levels, so that the secure cloud storage capability can be easily deployed to different applications without modifying or recompiling them. Saving data to the cloud and using the secret sharing scheme collectively ensure high data availability. SafeSky saves the data with their versions to the cloud and ***uses a simple saves-after-retrieval mechanism to correctly satisfy the single-reader single-writer consistency semantics***. The interception component updates the memory structures as soon as the application performs write operations, while multiple dedicated threads and reusable TCP connections are used by the data protection component and the cloud driver component to perform their tasks in a parallel and asynchronous manner, thus minimizing the perceivable performance overhead to end users. A SafeSky-enabled application does not need to save any data to the local disk, and the latest copy of the data can always be conveniently accessed from the cloud.

D. Interception Component

The interception component intercepts applications' file operations either at the standard C function level or at the system call wrapper function level using dynamic loading techniques. One widely used dynamic loading technique on the Linux platform is based on the LD_PRELOAD environment variable, which specifies other shared libraries that can be preloaded into an application's running process to take precedence over the original dynamically linked libraries used by the application. The functions implemented in the preloaded libraries will override the corresponding functions in the original libraries; therefore, the behavior of the application can be changed as desired without requiring any code modification or recompilation to the application.

1) Interception Strategy: The interception component intercepts both text and binary file operations at both the standard C function level and the system call wrapper function level. Table I lists the key intercepted file operations. If an application calls standard C functions, this component performs the interception at the standard C function level, and the operation will not be further passed down to the system call wrapper function level; otherwise, it performs the interception at the system call wrapper function level. This strategy allows SafeSky to flexibly support different applications on different platforms as discussed in Section III-C.

SafeSky allows developers or advanced users to specify the files, for which the data will be securely saved to the cloud, in the *Rules for Interception* configuration file as shown in Figure 1. One reason for using such a configurable mechanism is that in addition to users' data files, applications often write many temporary files which are not necessary to be saved to

TABLE I: Intercepted file operation functions

	File operations
Standard C function level	fopen(), fclose(), fread(), fwrite(), ...
System call wrapper function level	open(), open64(), creat(), creat64(), close(), read(), write(), lseek(), lseek64(), stat(), stat64(), lstat(), lstat64(), fstat(), fstat64(), ...

the cloud; the other reason is that users can have the flexibility to specify the files they want to save to the cloud.

2) Memory Structure and Interceptions: For each specified data file, SafeSky will maintain a block-level granularity memory structure that includes the folder name, file name, file open mode, read/write offset, file length, block size, and a table of data blocks as shown in Figure 2. The read/write offset is the file offset (very similar to the current active pointer in the FILE structure in C), and it is the start position for reading/writing data from/to a data block in the memory structure. Each data block contains the index, length, memory version, cloud version, and content information; the contents of all the data blocks in a memory structure constitute the content of the corresponding file accessed so far; the memory version records the current version number of a data block in the memory structure; the cloud version records the version number of a data block saved to the cloud.

In the file opening functions (e.g. open() and fopen()) implemented in SafeSky, once a specified file is opened, a memory structure is created. To present the same semantics to an application between using the file system and using the cloud storage, SafeSky supports frequently used file operation flags such as O_CREAT and O_APPEND as well as rarely used flags such as O_SYNC. In the file closing functions (e.g. close() and fclose()) implemented in SafeSky, once a specified file is closed, the newly created or updated data blocks are protected and uploaded to the cloud by the data protection component and the cloud driver component, respectively.

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
1  file_name = getNameFromFileDescriptor(fildes);
2  int ret;
3  if isSpecifiedInTheRuleFile(file_name) then
4      ret = writeMemoryStructure(file_name, buf, nbyte);
5      if isSynchronizedWrite(file_name) then
6          ret = sendDataSaveMessage();
7  else
8      ret = orig_write(fildes, buf, nbyte);
9  return ret;
```

Fig. 3: Pseudo code for the write() function

In the file writing functions (e.g. write() and fwrite()) implemented in SafeSky, the written data will be updated to the corresponding data blocks in the memory structure. Figure 3 illustrates the pseudo code for the write() system call wrapper function implemented in SafeSky. If the file is specified in the *Rules for Interception* configuration file, this function updates the memory structure and its data blocks at line 4, and sends a message to the data protection and cloud driver components to immediately protect and save the newly written data blocks to the cloud at line 6 if the file is opened with synchronized file operation flags such as O_SYNC; otherwise at line 8, it calls the original write() system call wrapper function, whose pointer $orig\_write$ was obtained through the system call *dlsym(RTLD_NEXT, "write")* when SafeSky was initialized.

The logic of the implemented file reading functions (e.g. read() and fread()) in SafeSky is similar to that of the file writing functions; however, the corresponding data blocks existing in the memory structure will be directly returned to the application, while nonexistent data blocks will be retrieved and recovered from the cloud.

Since SafeSky maintains each memory structure at the block-level, read and write operations performed on-demand by applications can be efficiently supported. Note that if the size of the memory structures becomes too large, the least-recently used (LRU) cache replacement algorithm can be used to evict some data blocks and free certain memory space.

E. Data Protection Component

When a data block needs to be saved to the cloud, its memory version, block index, block length, and block content together with the metadata such as file length and block size in the memory structure are extracted to form a plaintext. The small-size metadata is always bound to the data block in a plaintext so that its transmission and maintenance overhead could be minimized. This plaintext is first protected using an authenticated encryption algorithm (e.g. the NIST-approved CCM algorithm [10]) with a randomly generated key; the generated ciphertext along with the cipher type (AE-type), the parameters (AE-params), and the key (AE-key) used in the authenticated encryption are then supplied to the Shamir's (k, n) secret sharing scheme [17] with parameters N and K to produce N secret-shared data block pieces, each of which together with the parameters N, K, and the version (copied from the memory version) form a ***cloud data object***. Each cloud data object will be finally saved by the cloud driver component to a storage service, and indexed by an *id* generated from the hash of the folder name, file name, block index, and the identifier (e.g. domain name) of that cloud storage service.

In the decryption and verification process, any K cloud data objects of a data block can be used by the secret sharing scheme [17] to recover the ciphertext, which will be decrypted and verified using the authenticated decryption algorithm to reconstruct that data block.

The authenticated encryption algorithm is used to ensure both the confidentiality and integrity of the data blocks, so that it is computationally infeasible for attackers to decrypt the ciphertext, and any unauthorized modification to the cloud data objects can be detected. The secret sharing scheme is used to further strengthen the confidentiality and ensure the availability of the cloud data objects. In terms of the confidentiality, even if attackers can compromise any $K - 1$ cloud storage services and steal any $K - 1$ cloud data objects of a data block, it is ***absolutely infeasible*** [17] to recover the entire ciphertext of that data block, and further recover the corresponding plaintext due to the incomplete ciphertext. In terms of the availability, the entire ciphertext of the data block can be recovered from any K or more cloud data objects [17] retrieved from any K or more available cloud storage services. This availability guarantee can also help mitigate the cloud service vendor lock-in risks [1], [3].

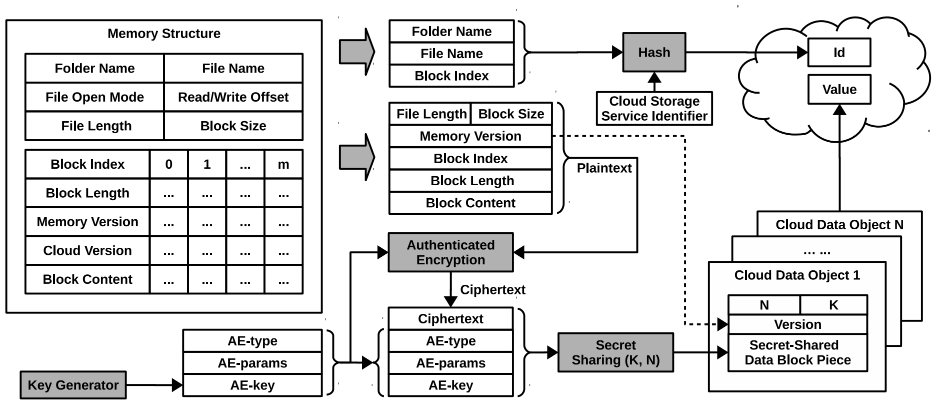


Fig. 2: Memory structure and data protection in SafeSky

It is worth mentioning that the cipher type, parameters, and key used in the authenticated encryption algorithm is also secret shared (Figure 2) so that they need not be locally saved or deterministically derived based on certain secret information provided by a user. Similarly, the parameters N and K used in the secret sharing scheme [17] are saved to the cloud along with each secret-shared data block piece so that they need not be locally saved or provided by a user.

F. *Cloud Driver Component*

The cloud driver component saves/retrieves the cloud data objects to/from different cloud storage services. As highlighted in Section II, popular cloud storage services offer free accounts and storage spaces to regular users, and they follow the predominant REST (Representational State Transfer) Web service design model to allow different client applications to easily access them. The cloud driver component needs to use the REST APIs of those cloud storage services to perform LIST, PUT, GET, and DELETE interactions with them. SafeSky simply uses the storage capability of cloud storage services, and it does not need any special computational support from them and does not require any modification to them. This design decision is important for SafeSky to easily support the use of different cloud storage services.

1) User Authentication: At the beginning of each application session, this component authenticates a user to the cloud storage services using the protected cloud accounts provided by the user. It supports common user authentication methods used by popular cloud storage services. One method is the traditional password based authentication, which is used by services such as Rackspace and Swift. Another method is the single sign-on authentication that uses access tokens for accessing services, and it becomes increasingly popular in recent years with the wide adoption of the OpenID [35] and OAuth [34] standards; for example, Dropbox, Box, Google Drive, Microsoft OneDrive, etc., all require client applications to use the OAuth 2.0 protocol to obtain access to their services.

Correspondingly, a user's cloud accounts can include both username/password pairs and single sign-on access or refresh tokens. Based on the user's preference, the cloud accounts can be protected either by the operating system (e.g. using the keyring mechanism on the Linux platform) or by using an additional master password supplied by the user. Based on the number of the provided cloud accounts, SafeSky can suggest the default values for the parameters N and K used in the secret sharing scheme [17], and advanced users can also modify the default values if they want.

2) Cloud Data Retrieval, Save, and Consistency: Within an application session, when a data block is read for the first time by the application, the cloud driver component retrieves K cloud data objects from any K of the N cloud storage services; this retrieval operation will only occur once in a session for each data block of each file. SafeSky creates a separate master thread to periodically inspects the memory structures for all the files. In a memory structure (Figure 2), if the memory version of a data block is newer than its cloud version, this master thread wakes up an idle worker thread in a thread pool to instruct the data protection component for protecting the corresponding plaintext, and instruct the cloud driver component with a pool of reusable TCP connections for saving the N cloud data objects to N different cloud storage services. When the application closes files or its session ends, the master thread also examines the memory structures to see if some final protection and save operations are needed.

Such a saves-after-retrieval consistency mechanism is simple and appropriate for single-user applications, which are most widely used and they only need to satisfy the single-reader single-writer consistency semantics. Once a memory structure is constructed from the cloud data objects retrieved from the cloud for a file, no more data will be retrieved from the cloud to replace the data blocks in the memory structure; it can only be further updated by the interception component of SafeSky based on the application's write operations. By using a separate master thread, a pool of worker threads, and a pool of reusable TCP connections, and by using the saves-after-retrieval consistency mechanism, the data protection component and the cloud driver component perform their tasks in a parallel and asynchronous manner for reducing the perceivable performance overhead to end users. In addition, the cloud driver component contains a cache, which can hold the data prefetched from the cloud and potentially further reduce the perceivable performance overhead to end users.

Such a simple design also allows us to correctly meet the consistency requirement of SafeSky. SafeSky requires that the value of K must be greater than a half of that of N. A successful save operation requires that the freshest version of at least K cloud data objects of a data block are successfully uploaded to K available cloud storage services; the freshest version number is copied from the memory version as shown in Figure 2. SafeSky uses the majority consensus solution [4], [19] to identify the freshest version number of the retrieved cloud data objects; a successful retrieval operation requires that the freshest version of at least K cloud data objects of a data block are successfully retrieved from K available cloud storage services. SafeSky will perform retries for failed operations with the assumption that at least K cloud storage services are available at any time in an application session.

IV. Implementation

We implemented SafeSky as a C shared library on a Ubuntu Linux system. It supports applications written in languages such as C, Java, and Python as long as they interact with the underlying operating system through the dynamically linked GNU *libc* library, which is used as the C library in the GNU systems and most systems with the Linux kernel [38]. It supports popular cloud storage services such as Amazon Cloud Drive, Box, Dropbox, Google Drive, Microsoft OneDrive, Rackspace, and Swift; it supports both password and single sign-on user authentication methods used by those services.

In the implementation of the data protection component, we used the *libcrypto* library for authenticated encryption and decryption, and used the *libgfshare* library for Shamir's (k, n) secret sharing scheme [17]. In the implementation of the cloud driver component, we used the *libcurl* library for user authentication and REST API interactions with cloud storage services, and used the *libjson* library for parsing the received responses from cloud storage services. All these four libraries are provided by default on Linux systems such as Ubuntu. The total number of lines of code in SafeSky is about 6,300.

V. Evaluation

We used three free and full-blown applications, HomeBank, SciTE Text Editor, and Firefox Web browser, from the Ubuntu Software Center to evaluate SafeSky. HomeBank [29] can assist users in managing their personal accounting. It has many analysis and graphical representation features, and can use different types of files to save users' personal accounting information. SciTE Text Editor [36] is similar to most text editors. It has additional features such as automatic syntax styling and can partially understand the error messages produced by many programing languages. Firefox is a popular Web browser that saves many types of users' browsing data such as bookmarks, history records, cookies, form values, and website passwords. These three applications cover both text and binary file operations at both the standard C function level and the system call wrapper function level.

We used four cloud storage services, Dropbox [27], Box [26], and two Swift [37] services deployed on two Amazon EC2 instances. Dropbox and Box use OAuth based user authentication, while Swift uses password based user authentication. The two Swift services are located at the east coast and the west coast, respectively, to purposefully consider geolocation diversity in our performance evaluation. We used four as the value for both parameters N and K in the Shamir's (k, n) secret sharing scheme [17]; the value of K is maximum so that we can measure and report the worst case performance in our evaluation, while in the real use of SafeSky the value of K can often be less than that of N as we also tested.

We evaluated the correctness and performance of SafeSky on a computer with 3.4GHz CPU and 8 GiB memory. We ran the experiments 10 times and present the average results. We have not done a usability study for SafeSky yet because we focus on its feasibility in this paper.

A. Correctness

We intensively and manually experimented with the file operation related features of the three applications to examine if SafeSky has been seamlessly loaded into them. We verified that the three applications worked properly as usual, while users' data are saved to the cloud rather than to the local disk. In the interception component, SafeSky correctly intercepted all the file operations, created and maintained memory structures, and returned data to applications. In the data protection component, SafeSky correctly performed the authenticated encryption, authenticated decryption, and secret sharing operations. In the cloud driver component, SafeSky correctly performed user authentication, data save, and data retrieval operations with the four cloud storage services.

B. Performance

We automatically evaluated the memory structure maintenance and cryptographic operation performance, evaluated the cloud data save and retrieval latencies, and measured the data block read and write frequency of the applications.

1) Memory Structure Maintenance: We compared the time for reading/writing data from/to a memory structure (i.e., with the using of SafeSky by the applications) with the time for reading/writing the same data from/to a local disk file (i.e., without the using of SafeSky by the applications). Overall, the memory structure maintenance performed by SafeSky in a read or write interception does add small additional performance overhead due to the memory allocation and memory copy operations. However, the overhead is negligible and only at the microsecond level.

2) Cryptographic Operations: Figure 4 illustrates the performance of the authenticated encryption, authenticated decryption, secret sharing encryption, and secret sharing decryption operations. As the data size increases from 2KB to 64KB, both the AES-CCM [10] encryption time and decryption time remain small within one millisecond. The secret sharing encryption time and decryption time increase linearly with the increase in data size, and the encryption always takes more time than the decryption. Because decryption operations are performed by SafeSky only once in an application session for each data block and encryption operations are periodically performed in separate worker threads, their performance overhead is not a big concern for the overall application session.

3) Data Save and Retrieval Latencies: We evaluated the data save and retrieval latencies on those four cloud storage services. The save latency for a certain number of data blocks is the time used by SafeSky to successfully PUT all the corresponding cloud data objects to those four cloud storage services. The retrieval latency for a certain number of data blocks is the time used by SafeSky to successfully GET all the corresponding cloud data objects from those four cloud storage services.

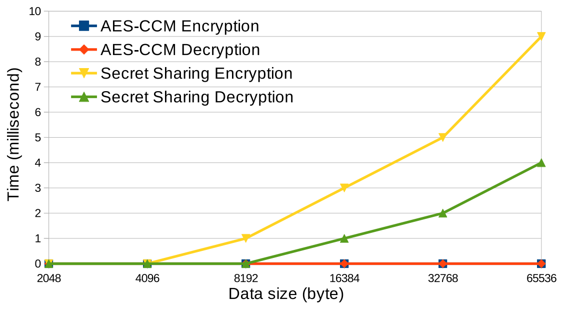


Fig. 4: Cryptographic operation performance

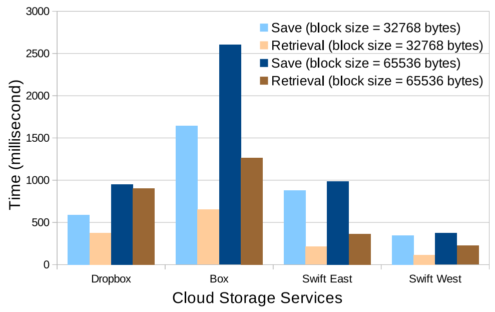


Fig. 6: Single data block save and retrieval latencies

Figure 5 illustrates the experimentally measured worst-case save and retrieval latencies for files with different sizes. The worst-case save latency is incurred when all the data blocks of a file are updated by an application in a short period of time and thus need to be saved to the cloud. The worst-case retrieval latency is incurred when all the data blocks of a file are read together for the first time by an application and thus need to be retrieved from the cloud. We measured the worst-case save and retrieval latencies of five files with sizes increased from 3.2768 MB to 16.384 MB. We experimented with two data block sizes 32,768 bytes and 65,536 bytes; correspondingly, the total number of data blocks in those five files increases from 100 to 500 for the 32,768-byte data block size, and from 50 to 250 for the 65,536-byte data block size. The 32,768-byte data block size is suggested in BlueSky because a smaller block size such as 4,096-byte will incur higher performance overhead for a system that relies upon wide-area transfers [20]. We used the 65,536-byte data block size to measure if a larger block size could further reduce the worst-case save and retrieval latencies.

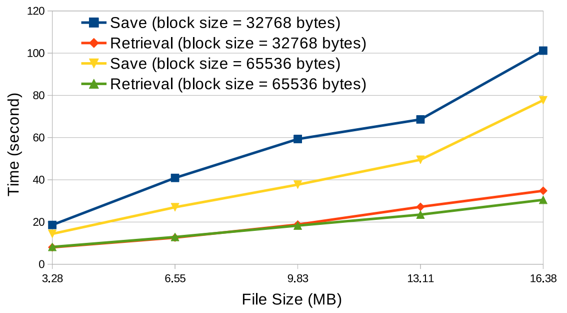


Fig. 5: Measured worst-case file save and retrieval latencies

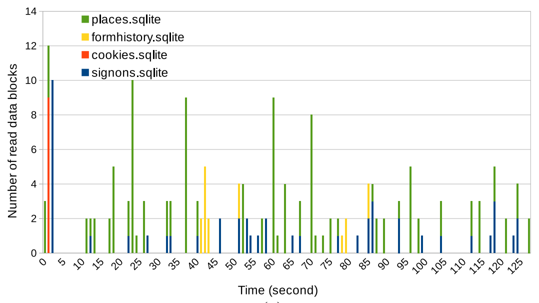
Basically, with the increase of the file size, both the worst-case save latency and retrieval latency increase; meanwhile, the save latency is always larger than the retrieval latency. With the same file size, the worst-case retrieval latency for the 65,536-byte data block size is slightly smaller than that for the 32,768-byte data block size, while the worst-case save latency for the 65,536-byte data block size is about one third smaller than that for the 32,768-byte data block size.

A larger data block size can help reduce the worst-case save and retrieval latencies. However, applications usually read and write a portion of a file on-demand each time, corresponding to a single or a handful of data blocks; therefore, considering the save and retrieval latencies for a single block is often more important than considering the worst-case save and retrieval latencies. Figure 6 illustrates the single data block save and retrieval latencies; it shows that the 32,768-byte data block size outperforms the 65,536-byte data block size on both save and retrieval for all the four cloud storage services. Collectively, we suggest that the 32,768-byte data block size should be used in SafeSky to efficiently support most applications. However, it a user frequently opens an application, the repeated initial retrieval operations may still cause perceivable delays to the user.

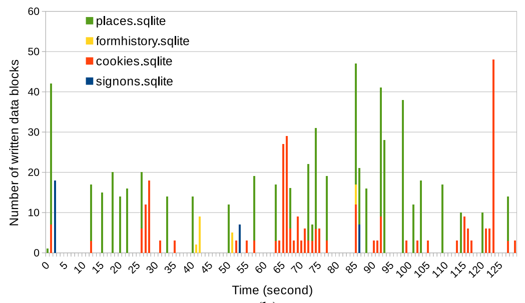
4) Data Block Read and Write Frequency: Different applications have their unique data block read and write frequency patterns, depending on how a user and an application use the files. In HomeBank and SciTE Text Editor, one or more data block read operations will be performed when a user opens a file, and one or more data block write operations will be performed when a user saves the records or the edited text to a file. In Firefox, a user's browsing data are saved to multiple SQLite database files; when a user performs browsing tasks, data block read and write operations will be triggered by Firefox to the corresponding database files. Because the file operations performed by Firefox are more intensive and diverse than those of the other two applications, we measured the data block read and write frequency of Firefox with the 32,768-byte data block size to demonstrate that SafeSky is capable of handling the intensive file operations performed by complex end-user applications.

We designed a browsing session scenario consisting of seven main steps. Step 1, we visit the Google homepage, add it to bookmarks, perform a search using the keyword "security", and click one link on the response page. Step 2, we visit the CNN homepage and add it to bookmarks. Step 3, we visit the Facebook login page, add it to bookmarks, log into it, allow Firefox to remember the login password, and log out. Step 4, we visit the Fox News homepage and add it to bookmarks. Step 5, we visit the Gmail login page, add it to bookmarks, log into it, allow Firefox to remember the login password, and log out. Step 6, we visit the YouTube homepage, add it to bookmarks, and click the link to one video. Step 7, we revisit all those six webpages from their bookmarks, and let Firefox autofill the login forms on the Facebook and Gmail login pages.

We performed this browsing session scenario quickly in approximately two minutes to intensively trigger the file operations of Firefox. During the browsing session, Firefox reads/writes bookmark records and history records from/to the *places.sqlite* database file, reads/writes name and value pairs of form fields from/to the *formhistory.sqlite* database file, reads/writes website cookies from/to the *cookies.sqlite* database file, and reads/writes login passwords from/to the *signons.sqlite* database file.



(a)



(b)

Fig. 7: Data block (a) read and (b) write frequency in a browsing session

Figures 7(a) and 7(b) illustrate the data block read frequency and write frequency of those four database files in our browsing session, respectively. These are the results for just one browsing session; averaging the results from multiple runs does not make sense because file operation characteristics is unique for every browsing session. Read operations on the *places.sqlite* and *signons.sqlite* database files occurred most frequently because bookmark, history, and form field records are frequently examined by Firefox on each webpage. Read operations on the *formhistory.sqlite* database file occurred only for webpages that contain forms. Read operations on the *cookies.sqlite* database file occurred only at the beginning of the browsing session; we conjecture that this phenomenon is due to the possible reason that Firefox caches all the cookies in memory at the beginning of a browsing session, so that the intensive use of cookies in almost every webpage will not incur too much performance overhead. Correspondingly, we observed that write operations on the *places.sqlite* and *cookies.sqlite* database files occurred most frequently, while write operations on the *formhistory.sqlite* and *signons.sqlite* database files only occurred for those two login webpages.

Both data block read and write operations are intensively performed in this browsing session experiment. However, because all the read operations are served by SafeSky using the data blocks managed in the memory structures for the corresponding files, and all the write operations are performed to the memory structures while separate worker threads are used to save data to the cloud, such intensive and complex read and write file operations from Firefox can still be smoothly processed by SafeSky. We did not perceive any performance overhead in this browsing session. These performance evaluation results demonstrate that SafeSky can efficiently perform its functionality and can meet its performance requirement.

VI. Security Analysis

As analyzed in Section III-E, SafeSky first applies the authenticated encryption to ensure that it is computationally infeasible for attackers to break the data confidentiality and integrity. Furthermore, it applies the Shamir's (k, n) secret sharing scheme [17] to ensure: (1) it is absolutely infeasible for attackers to break the data confidentiality, given that they do not compromise k or more cloud storage services at the same time; (2) a high level of data availability can be achieved, given that any k cloud storage services are available to a user. A user's cloud data objects could still be obtained by unauthorized parties from k or more cloud storage services in highly rare situations, for example, due to simultaneous data breaches in k cloud storage services, the collusion of k cloud storage service providers, or the government surveillance; furthermore, by identifying the k corresponding cloud data objects of a data block, the unauthorized parties can compromise the confidentiality of that data block. However, SafeSky makes such an identification difficult by uniquely generating the *ids* of cloud data objects from a hash function salted with the storage service identifiers (Figure 2), albeit the timing information of block interactions may still be used by attackers.

At the user-end, if malware exists on a user's computer, the plaintext data, the cryptographic keys, and cloud accounts could be directly stolen from the memory to compromise the data confidentiality. Such potential attacks are out of the scope of this paper because they pose common risks to all the applications and data on a computer. However, users should still pay serious attention to the risks of malware and should immediately address the malware problem by either cleaning up or reinstalling the system.

As described in Section III-F, a user's cloud accounts can be protected either by the operating system or by using an additional master password supplied by the user. It is possible that the protected cloud accounts may be damaged or lost, for example, due to the crashing of the file system or the careless deletion by the user. However, in such cases, SafeSky ensures that the user's data can still be available; the user can simply use password reset mechanisms provided by the cloud storage services to regain cloud accounts, and then retrieve the cloud data objects to completely recover their data.

VII. Discussion

SafeSky supports user authentication and data save/retrieval operations on multiple cloud storage services such as Amazon Cloud Drive [25], Box [26], Dropbox [27], Google Drive [28], Microsoft OneDrive [33], and Swift [37]. Note that some of these services are not free for using their REST APIs and storage by client applications. For example, the costs of using Amazon Cloud Drive [25] and Google APIs Console in Google Drive [28] are both based on the storage size and network traffic. Users can have their own choices to select cloud storage services based on their preference and budget. For example, regular users can select free cloud storage services such as Box [26] and Dropbox [27], enterprise users may select paid cloud storage services with larger storage capability, and advanced users may set up their own storage services using software such as Swift [37]. Further reducing the cost of using cloud storage services that are not free is out of the scope of this paper, and we refer readers to the BlueSky paper [20] for more information.

Currently, SafeSky focuses on satisfying a simple single-reader single-writer consistency semantics for single-user applications that are most widely used, thus a saves-after-retrieval mechanism is sufficient. Satisfying a more general single-writer multi-readers consistency semantics is feasible by letting readers periodically check cloud storage services to retrieve fresher cloud data objects. Some collaborative applications allow multiple users to work on a common task simultaneously, and they require a more complex multi-reader multi-writer consistency semantics; however, supporting this consistency semantics by a solution such as our SafeSky is very difficult if not impossible because SafeSky simply uses the storage capability of cloud storage services without requiring any special computational support from them or any modification to them. In addition, currently SafeSky does not support the memory mapping operations such as mmap() and network operations such as send() because it cannot ascertain and may compromise the semantics of those operations.

VIII. Conclusion

In this paper, we took a middleware approach and designed SafeSky, a secure cloud storage middleware that can immediately enable either legacy or new end-user applications to have the secure cloud storage capability without requiring any code modification or recompilation to them. A SafeSky-enabled application does not need to save any data to the local disk, but instead securely saves the data to multiple free cloud storage services to simultaneously enhance the data confidentiality, integrity, and availability. We implemented SafeSky as a C shared library on Linux. SafeSky supports applications written in different languages, various popular cloud storage services, and common user authentication methods used by cloud storage services. We evaluated the correctness and performance of SafeSky by using real-world applications and analyzed its security. Our evaluation and analysis results demonstrate that SafeSky is a feasible and practical approach for equipping end-user applications with the secure cloud storage capability.

Acknowledgment

We sincerely thank anonymous reviewers for their valuable suggestions and comments. The first two authors were previously at UCCS, and they were supported in part by NSF grants CNS-1359542 and DGE-1438935.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proc. of the ACM symposium on Cloud Computing (SoCC)*, pages 229–240, 2010.

[2] G. Anthes. Security in the cloud. *Commun. ACM*, 53(11):16–18, 2010.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[4] P. A. Bernstein. Getting consensus for data replication: Technical perspective. *Commun. ACM*, 57(8):92–92, 2014.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. In *Proc. of the EuroSys*, pages 31–46, 2011.

[6] M. Blaze. A cryptographic file system for unix. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 1993.

[7] K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 187–198, 2009.

[8] B. Calder, J. Wang, A. Ogus, N. Nilakantan, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[9] W. Du, K. Jayaraman, X. Tan, T. Luo, and S. Chapin. Position Paper: Why Are There So Many Vulnerabilities in Web Applications? In *Proc. of the New Security Paradigms Workshop (NSPW)*, 2011.

[10] M. Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. In *NIST Special Publication 800-38C*.

[11] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[12] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proc. of the USENIX Security Symposium*, 2012.

[13] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Proc. of the Financial Cryptography (FC)*, pages 136–149, 2010.

[14] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.*, 29(4):1–38, 2011.

[15] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big" web services: making the right architectural decision. In *Proc. of the International Conference on World Wide Web (WWW)*, 2008.

[16] R. A. Popa, J. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. of the USENIX Annual Technical Conference*, 2011.

[17] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[18] H. Takabi, J. B. D. Joshi, and G.-J. Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy*, 8(6):24–31, 2010.

[19] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[20] 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.

[21] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou. Toward Secure and Dependable Storage Services in Cloud Computing. *IEEE Trans. Serv. Comput.*, 5(2):220–232, 2012.

[22] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proc. of the Annual USENIX Technical Conference*, pages 197–210, 2003.

[23] C. Yue. Toward Secure and Convenient Browsing Data Management in the Cloud. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[24] R. Zhao, C. Yue, and Q. Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proc. of the International Conference on World Wide Web (WWW)*, pages 1384–1394, 2015.

[25] Amazon Cloud Drive. http://www.amazon.com/gp/feature.html?ie=UTF8&docId=1000828861.

[26] Box Cloud Storage. https://www.box.com/.

[27] Dropbox Cloud Storage. https://www.dropbox.com/.

[28] Google Drive. https://drive.google.com/.

[29] HomeBank. http://homebank.free.fr.

[30] HP Cloud Object Storage. https://www.hpcloud.com/products/object-storage.

[31] iCloud. http://www.apple.com/icloud/.

[32] iCloud Data Breach. http://www.forbes.com/sites/davelewis/2014/09/02/icloud-data-breach-hacking-and-nude-celebrity-photos/.

[33] Microsoft OneDrive. http://windows.microsoft.com/en-us/onedrive/skydrive-to-onedrive.

[34] OAuth 2.0 Authorization Framework. http://tools.ietf.org/html/rfc6749.

[35] OpenID 2.0. http://openid.net/specs/openid-authentication-2_0.html.

[36] SciTE: a SCIntilla based Text Editor. http://scintilla.org/SciTE.html.

[37] Swift - OpenStack. https://wiki.openstack.org/wiki/Swift.

[38] The GNU C Library. http://www.gnu.org/software/libc/libc.html.