Thomas Lenz, Bernd Zwattendorfer, Arne Tauber:"A Secure and Confidential Javascript Crypto-Framework for Cloud Storage Applications", IADIS International Conference WWW/INTERNET, 2013, pp. 219-226

A SECURE AND CONFIDENTIAL JAVASCRIPT CRYPTO-FRAMEWORK FOR CLOUD STORAGE APPLICATIONS

Thomas Lenz, Bernd Zwattendorfer and Arne Tauber
E-Government Innovation Center (EGIZ)
{thomas.lenz, bernd.zwattendorfer, arne.tauber}@egiz.gv.at

ABSTRACT

Cloud computing is currently one of the fastest growing segments in the ICT sector. Although cloud computing has many benefits, still some issues, particularly relating to security and privacy, can be identified and must not be ignored. Especially, if sensitive or personal data wants to be stored in the cloud, these data has to be highly protected from unauthorized access. To avoid unauthorized access, encryption can be used to keep the data confidential. Various approaches and implementations using encryption techniques to keep data confidential exist. However, all of them require some special client software, which has to be installed on the user's local machine. To avoid such local binary installations, we propose a fully browser-based solution to encrypt data and store it securely in the cloud. The proposed solution relies on existing and already implemented web browser technologies, only. By the help of our solution, users can simply drag and drop files into their browser window to store them securely in the cloud. The proposed secure and confidential remote data-storage application is fully based on a JavaScript Crypto-Framework, which in turn relies on approved and wide-spread cryptographic standards for symmetric and asymmetric encryption. While the use of our solution provides a lot of benefits to users, we could also identify some limitations in terms of security. In a security analysis we discuss some general aspects of JavaScript cryptography, evaluate the security of our web-based encryption tool in detail, and introduce some measures to improve the security of browser-based cryptography.

KEYWORDS

JavaScript, Cloud Computing, Cryptography, Security, Confidentiality, Encryption

1. INTRODUCTION

Cloud computing gained a lot of popularity over the past years and is currently still one of the dominant topics in the ICT sector. Its advantages such as nearly unlimited IT resources and its pay-as-you-go model promise many benefits and an enormous cost savings potential to its customers. However, although cloud computing offers a lot of benefits, still some issues and challenges must be met and cannot be ignored when applying cloud computing. Some of the biggest issues hindering an increasing adoption of cloud computing relate to security and privacy concerns (Kandukuri et al., 2009; Popovic and Hocenski, 2010).

Cloud computing is currently heavily used for data storage, e.g. for archiving or backup purposes. Popular examples for such cloud services are Drop-Box[1] , Google Drive[2] , or Microsoft SkyDrive[3] . These services offer cloud storage and file synchronization between various clients at the same time. While non-sensitive data can be easily moved or transferred to such cloud providers, security concerns may arise if sensitive or personal data need to be transferred. Usually, sensitive data must be especially protected from unauthorized access due to various reasons (e.g. national policies, regulations, or even legislations). These reasons particularly take effect if data are moved into the cloud because the cloud provider may have access to those data any time if it is provided unencrypted.

To avoid such unauthorized and unwanted access by the cloud service providers or third parties, data should be encrypted before moving it into the cloud to keep it confidential. Hence, without having the appropriate encryption key the cloud service provider will not be able to inspect or investigate the stored

[1] https:// www.dropbox.com/
[2] https://drive.google.com/
[3] https://skydrive.live.com/

data. The idea of encrypting data before moving it to the cloud is not new, thus several approaches exist. The most promising and most popular solutions apply encryption and decryption functions on client side before transferring the data to the cloud. However, currently all of those solutions require binary client software or a special browser plug-in to be installed on the user's local machine. To bypass this requirement, we propose a solution that allows client-side data encryption for cloud storage without installing separate software modules or browser plug-ins. Achieving this, in our solution we fully rely on browser-based technologies such as JavaScript or HTML5 (W3C, 2012), which are supported by most popular web browsers.

Our paper is structured as follows. Section 2 overviews related work on secure cloud storage and briefly discuss advantages and disadvantages of existing solutions. In Section 3 we describe our proposed solution based on JavaScript cryptography and give details on our implementation. In Section 4, we analyze security aspects of JavaScript cryptography in general and particularly relating to our solution. Finally, we draw conclusions in Section 5.

2. RELATED WORK

Keeping data confidential and protected from unauthorized access does not define a new desire. Several approaches to keep data confidential in local or remote systems do exist. Usually, the security requirement on confidentiality is met by using data encryption-techniques. In this section, we briefly describe different approaches to keep or store data securely and confidentially in remote systems such as the cloud by relying on data encryption-techniques.

Currently, the most common approaches guaranteeing confidentiality are simple container formats, which allow for encryption of the contained data. A popular example for a container standard is S/MIME (Secure/Multipurpose Internet Mail Extensions) (Ramsdell and Turner, 2004), which is mostly used for encrypting e-mails. Citizen Card Encrypted (CCE) (Teufl and Suzic, 2012), for instance, is a tool which relies on S/MIME and enables encryption and decryption of arbitrary data using the Austrian citizen card (Leitold et al., 2002), the official electronic identity (eID) in Austria. Another popular and widely spread software for encrypting data is TrueCrypt[4]. However, still many other tools supporting encrypted containers exist. Although all of these approaches focus on data encryption in local systems, they still can be used for storing data confidentially in remote systems, e.g. the cloud. In this case, data are encrypted locally within a container and are transferred to a remote server. Nevertheless, the problem is that files are not synced immediately with the remote system after encryption but instead require the encryption process of the whole container or data volume to be finished before transfer.

Especially the increasing popularity of cloud computing has brought up new solutions in this area. Confidentiality of data in the cloud is particular essential if sensitive data are stored, which should not be accessible by the cloud service provider. Different approaches for protecting and encrypting data in the cloud exist. One example is a middleware approach, where data is encrypted by a middleware before transferring the data into the cloud. There are various solutions which base on this middleware approach. Such solutions are described by (Diallo et al., 2012) or (Seiger et al., 2011) However, also the famous and popular cloud service DropBox relies on this approach and acts as some kind of middleware (Dropbox, 2013). Nevertheless, these middleware approaches still have the disadvantage that the middleware manages the encryption keys and thus always has the possibility to decrypt the data.

More sophisticated and higher confidential services rely on user-centric approaches, which apply encryption mechanisms on client side. In this case, data are encrypted on the user's client before sending it to the cloud provider. The main advantage is that the encryption keys remain in possession and under sole control of the user and the cloud service provider has no possibility to decrypt any data. Compared to the container-based approach, encrypted files sync automatically with the cloud service and are copied or transferred on the fly. Implementations of this approach are e.g. Wuala[5], BoxCryptor[6] or SpiderOak[7]. However, all of these implementations require special client software installed on the users local machine or in the users web browser.

[4] http://www.truecrypt.org/
[5] http://www.wuala.com/
[6] https://www.boxcryptor.com/
7 https://spideroak.com/

3. BROWSER-BASED ENCRYPTION TOOL

In this section, we present our solution of a browser-based encryption application, which can be used for file encryption and decryption. All cryptographic operations are performed within the browser on the user's local machine by relying on JavaScript and HTML5 features only. The first sub-section contains a description of the architecture of our solution. In the second sub-section, we explain the process flow, the cryptographic operations, and the data model, which we use. The last sub-section includes details on the implementation of our browser-based encryption tool.

3.1 Architecture

Figure 1 illustrates the general architecture of our browser-based encryption tool. The key component of the entire solution is the JavaScript Crypto-Framework, which provides all functionality to encrypt or decrypt data. The main part of this Framework is the Process Flow Engine, which coordinates the different steps of a file encryption or decryption process. All operations, which are required for the encryption or decryption process, are encapsulated in two sub modules, the Key Management module and the Crypto Engine module.

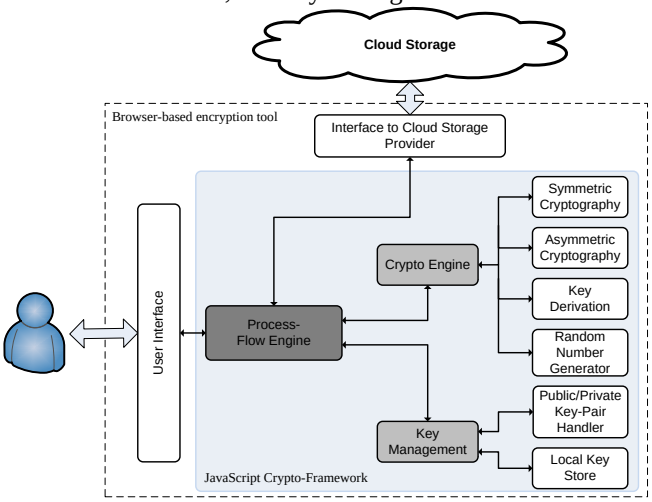

Figure 1. Architecture of the browser-based encryption tool.

The Key Management sub module implements several functionalities to handle the public/private key-pair processing (Public/Private Key-Pair Handler) and the access to a Local Key Store. This Local Key Store is used to store public/private key pairs, which are frequently required. The Crypto Engine includes all algorithms, which are needed for Symmetric and Asymmetric Cryptography, for Key Derivation, and secure random-number generation (Random Number Generator).

The browser-based encryption tool provides two interfaces. The first interface is the User Interface, which is implemented in HTML. This User Interface provides all functionality for key management, file handling, and file encryption and decryption procedures. The second interface is the interface to the cloud storage provider (Interface to Cloud Storage Provider). This interface is used to exchange encrypted data between the browser-based encryption tool and the cloud storage. We actually use a HTTP POST (Fielding et al., 1998) request method to transmit a single encrypted file to our own cloud storage provider. The usage of other storage providers, like Dropbox for example, is also possible with our architecture, because only the actually used interface has to be changed to the DropBox Core API[8].

3.2 Process Flow

The process flow of the data encryption-operation, as example consists of three steps. At first, the user has to select the public key, which should be used to perform the encryption. In the second step, a user can

[8] https://www.dropbox.com/developers/core/

select several files which should get encrypted and securely stored on a remote server. At last, the cryptographic encryption-operations are performed on the selected files and finally the encrypted files are uploaded to the cloud storage.

Different solutions how a file or data can be encrypted exist. Such solutions can be based on public-key cryptosystems, symmetric-key cryptosystems, or hybrid cryptosystems. An advantage of a public-key cryptosystem is that sender and receiver do not have to share a common secret in order to communicate securely as it is required in symmetric-key cryptosystems. A disadvantage of public-key cryptosystems is that they often rely on complex mathematical computations. I.e. in case of a large file size an encryption or decryption operation takes a long time. In comparison to that, symmetric-key cryptosystems are much faster. To combine the advantages of both cryptosystems, our JavaScript Crypto-Framework uses the hybrid cryptosystem approach. This cryptosystem approach combines the convenience of a public-key cryptosystem with the efficiency of a symmetric-key cryptosystem.

In the following we illustrate the usage of our JavaScript Crypto-Framework by describing a file encryption process in detail.

3.2.1 File Encryption

Figure 2 shows the process flow of a file encryption-operation. This process flow consists of several steps. A new symmetric key is generated at first. Therefore, we use the random-number generator which is part of the Stanford JavaScript Crypto Library (Stark et al., 2009).

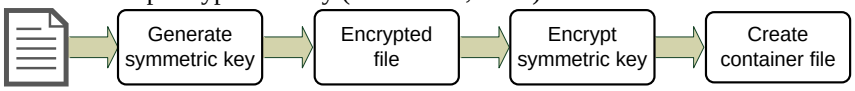

Figure 2. Process flow of an encryption operation.

In the second step, the file is encrypted by using a symmetric-key algorithm. Our solution uses an AES block cipher (National Institute and Technology, 2001; Stark et al., 2009) with a key size of 128 bits to perform the encryption of one 128-bit block. For file encryption, the AES block cipher is used in CCM mode of operation (Whiting et al., 2003).

After finishing the file encryption, the symmetric key is encrypted by using a public-key algorithm. This second encryption step is necessary, because the symmetric key is part of the file container which is uploaded to the cloud storage. By using a public-key algorithm, only the user who owns the corresponding private key is able to decrypt the symmetric-key information. We use the public-key algorithm RSA to perform the encryption of the symmetric key. Therefore, a public/private key pair is required, whereas our solution can handle a key size from 512 bits to 4.096 bits. The Public/Private Key-Pair Handler implements the parsing of public and private-key information.

A file container is created in the last step of the file encryption-process, which consists of the encrypted file and the encrypted symmetric key. We use the JSON format to build this container, because JSON is derived from the JavaScript scripting language and hence it is an ideal choice for representing simple data structures.

An additional feature in our JavaScript Crypto-Framework is the Local Key Store. This Local Key Store can be used to manage private-key and public-key information. The key information can be stored in the Web browser by using the Web Storage API (Hickson, 2009a), which is standardized by the WorldWideWeb Consortium (W3C). Unlike cookies, the Web Storage falls exclusively under the purview of the client-side scripting and a storage capacity greater than 5 MB is recommended[9]. Access to this storage is restricted by the Same-Origin Policy (Barth, 2011) of the web browser. This policy is a security feature of the web browser and permits only the very server to read the data which has also stored the data. In addition to this security feature, we encrypt all key information, which are stored in the browsers Web Storage. The cryptographic key for encrypting the key information is generated by using a key derivation function. In our application we use the password-based key derivation faction (PBKDF2) (Kaliski, 2000) to generate a cryptographic key from a password. This means, users are only able to access the key information in the Web Storage if they know the correct password.

[9] http://www.w3.org/TR/webstorage/#disk-space

3.3 Implementation

We have implemented the architecture, shown in Figure 1 to demonstrate the applicability of our JavaScript Crypto-Framework. To handle files in a secure way, the user has to pick some cryptographic keys which should be used. Our application supports two different possibilities how the keys are managed. The first possibility is a one-time installation of the key store. In this case, no key information is permanently stored in the local web browser and the public/private key pairs are only available for the duration of the current session. Such a one-time key store is useful in case of using a public computer. The second possibility is the usage of a long-term key store, which permanently stores the keys in encrypted form within the local web browser. To store keys securely in the key store, the user has to use a password which is used to encrypt the key information.

Figure 3 illustrates the web front-end after the key store has been opened. All operations, like file upload or key management, can be carried out with simple drag and drop operations. Before an encryption and upload operation can be performed, the user has to insert or to select the public key, which should be used for this operation. If a public key is selected, then an arbitrary number of files can be dropped into the file-upload area. For each file which is dropped, an encryption and upload operation is started. This operation uses the Crypto Engine module to encrypt the file, correspondent to the process flow illustrated in Figure 2. After the encryption operation, the file is transmitted to the cloud storage by using an XMLHttpRequest (W3C, 2011), which is a part of the browser API. If the file encryption and upload operation is completed, then the files are shown in the file upload area as a file list (see Figure 4).

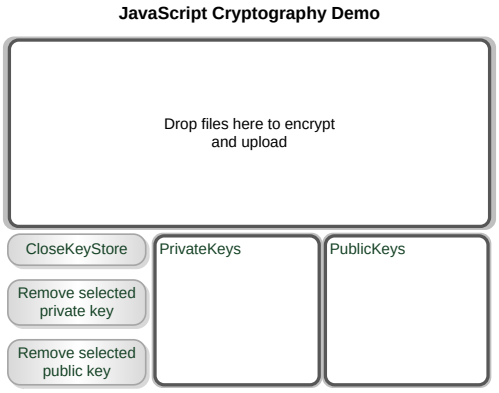

Figure 3. Web front-end with open key store and ready to use.

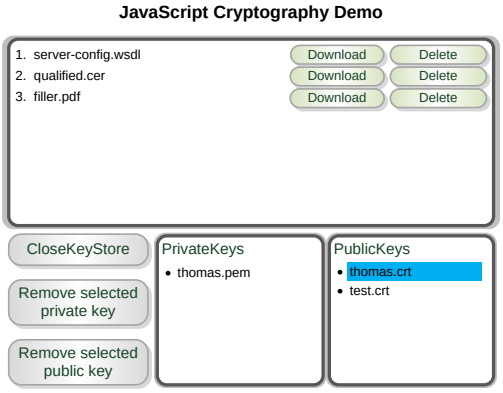

Figure 4. Web front-end with insert keys and uploaded files.

To download an encrypted file, the user has to choose the proper private key in the *PrivateKeys* area. Afterwards, the download and decryption operation can be started by using the *Download* button. In the case of bigger files, the whole encryption and upload / download operation can be time intensive. This may cause an unresponsive web page, because JavaScript is only single threaded. To prevent this unfavorable behavior we use the Web Workers (Hickson, 2009b) approach, which is specified by the W3C. This specification defines an API for spawning background scripts in a web application. Such a background script runs in an isolated thread and therefore the user interface is not blocked during execution.

4. SECURITY ANALYSIS

In the first sub-section, we discuss some general security aspects of JavaScript cryptography. In the second sub-section, we evaluate the security of our implemented browser-based encryption tool in detail and finally we present some solutions for security improvements.

4.1 General Security Aspects

JavaScript is a client-side scripting language, which can be used in almost all web browsers for complex applications, which need to be executed in the browser on client side. Although JavaScript is frequently used, the adoption of cryptographic operations in JavaScript has been mostly neglected so far. Nevertheless, Matasano Security published an article in the year 2010 dealing with JavaScript cryptography and its security (Matasano Security, 2010). They stipulated that JavaScript cryptography is doomed due to a number of factors, like secure delivery of JavaScript crypto libraries to the browser, random-number generation in a web browser, or the JavaScript view-source transparency. However, these criticisms are not valid for all applications in which JavaScript cryptography can be used.

The main criticism of the article written by Matasano is that client-side JavaScript cryptography cannot be used to substitute an SSL/TLS connection between the server and the local web browser. According to Matasano, it is not possible to make a secure challenge-response protocol to authenticate a user and initiate a secure channel by using only JavaScript on client side. Referring to him, the reason is that an attacker, who inspects the network traffic, can also manipulate the traffic and therefore the attacker can manipulate the cryptographic library, which is transmitted from the server to the client. Thus, a JavaScript only solution cannot be used to initialize a secure network-channel in an insecure network. However, if the HTTP connection is secured by using SSL/TLS, then a secure delivery of JavaScript libraries to the web browser is possible. In the case of our browser-based encryption tool, we want to store files securely in cloud storage and not protect the whole network channel. Additionally, most of the currently available web-browser applications support SSL/TLS by default. Consequently, an HTML application requires no additional software to protect the integrity of the JavaScript code from being changed by an outside adversary.

All JavaScript applications, which are executed in the local web browser, are vulnerable to attacks from the local machine, like malware or key-loggers. The reason is, that it is easy to inspect, debug, and manipulate the JavaScript application, which is executed in the web browser, because all modern web browsers already integrate additional programs for JavaScript debugging, like Firebug or the Debugger in Chrome. However, such attacks are generally a problem in software-based applications and hence are not a specific problem of client-side JavaScript applications.

The security of cryptographic operations, which are executed in the web browser, heavily depend on the strength of the cryptographic secure random-number generator (CSRNG). This may also represent vulnerability in JavaScript crypto applications because a predictable random-number generator cannot be used to generate secure cryptographic keys. Being still able to use secure random-number generation in JavaScript, CSRNG require a seed to initialize them. CSRNG can be seeded with entropy by e.g. using several browser events, like mouse-move events, key-information events, window size and position, history length, or clock information. Additionally, manually seeding can be used if the application asks the user to generate random mouse and keyboard inputs. This seed generation methods should be quickly enough to sustain the rate of random-number generations in our or in similar applications (Stark et al., 2009).

4.2 Solution-Specific Security Aspects

At first, we discuss the security of our browser-based encryption tool and identify some drawbacks in this scenario. Finally, we present different solutions, which could be used to bypass these drawbacks in future work.

4.2.1 Security of the Browser-based Encryption Tool

In a typical web application, the application provider provides all necessary JavaScript libraries. Consequently, this provider also delivers the cryptographic libraries. This circumstance comprises some security concerns. In comparison to an application provider who provides a plug-in or a locally installed binary application, which is downloaded and installed just once, a cloud-based application provider has to offer correct and secure cryptographic libraries every service delivery. Consequently, an application provider who delivers JavaScript libraries has to be treated as a trusted third party.

In case of our browser-based encryption tool, which is shown in Figure 1, some weaknesses arise during the JavaScript code delivery. The application provider and the cloud storage are from the same origin. In this

case, application provider and cloud storage could cooperate to attack an encryption operation when delivering a slightly different JavaScript library. Such a faulty and different JavaScript library need not be delivered permanently by an attacker, but could be delivered during particular events, like special times, special user operations or when interested file types are going to be transferred. If such an occasional attack is in use, then it makes it more difficult to detect dishonest cloud storage. Actually, there exists no solution to defend this security risk in such an application scenario. Consequently, besides the application provider it is necessary to trust the cloud storage too in our browser-based encryption tool. Such a server, providing the application and the cloud storage from the same origin, could be defined as an 'honest but curious' adversary, meaning the server performs its tasks correctly but may try to extract as much information as possible.

Actually, the WorldWideWeb Consortium (W3C) proposed a Web Cryptography API (W3C, 2013b), which could solve most of these security problems. The reason is that, if this API is implemented and included by a web browser, an application provider has to deliver the application without the cryptographic functionality only. Thus in this case, a web-based application relying on the W3C Crypto API has a similar level of security regarding the use of cryptographic functions as a plug-in or a local binary application.

4.2.2 Security Improvements

A weakness in the above-described architecture (see Figure 1) arises from the fact that application provider and cloud storage are in the same origin. In this case, we have to trust both service providers.

A security improvement can be achieved if the application provider is of another origin than the cloud storage. In this scenario, the browser-based encryption tool runs in the origin of the application provider and consequently, the cloud storage provider has no access to information in the sphere of the encryption tool. As a result, the trust assumption to the cloud storage is not required anymore, because the cloud storage is not in the same origin and the communication between the browser-based encryption tool and the cloud storage is through a well-defined interface (Interface to Cloud Storage Provider in Figure 1). All data which is transmitted via this interface, by using Cross-Origin Resource Sharing (W3C, 2013a), can be validated by the encryption tool. In addition, to avoid cross-site scripting (XSS) attacks we can use appropriate measures against through this interface.

Such a scenario is similar to the middleware approach mentioned in Section 2. However there are some differences. In particular, no local software – with the exception of the browser – is required on the user's machine. Taking DropBox as example, compared to its current implementation of the key management (Dropbox, 2013), which is handled by the remote DropBox servers, the key management can now be done by the application user on her local machine.

Another security constraint arises from the fact that a cloud-based application provider has to offer correct and secure cryptographic libraries every service delivery. This is a fundamental property of web-based applications. However, there are also some solutions to solve this weakness when using HTML5 applications. One solution is to use the encryption tool as HTML5 offline application. Thereby, the web browser is only used as an interpreter for a local available HTML5 application. In this case, the JavaScript libraries are under the full control of the user and the connection to the cloud storage is through a well-defined interface (Interface to Cloud Storage Provider in Figure 1). HTML5 offline applications have some advantages in contrast to binary ad-hoc software. Such HTML5 applications can be used on across platforms very easy, because web browsers with HTML5 support are standard on almost all platforms. Additionally, usage on restricted platforms is also simpler, because – with the exception of the browser – no other binary user application needs to be executed on such a platform.

Again, the best way to solve these security constraints would be the use of the W3C Web Cryptography API. Nevertheless, also in this case the browser has to be regarded as a trusted entity, but with the difference that the cryptographic libraries do not need to be transferred each time of application access.

5. CONCLUSION

Storing data or files confidentially on a remote server constitutes a frequent use case. Especially if cloud computing comes into play, storing data in the cloud without any possibility for the cloud service provider to inspect the data is vital. To keep and store data confidential in the cloud, usually encryption techniques are applied. Several tools already exist, which support data encryption and remote data transfer. However, all of

those tools require some special client software to be installed on the user's local machine or in the user's web browser.

To bypass such a requirement, we proposed a solution for secure and confidential remote data storage fully based on a JavaScript Crypto-Framework. By the help of this framework, data or files can be simply stored encrypted on a remote server or in the cloud using a standard web browser. Our solution just relies on existing and already implemented web browser technologies such as JavaScript or HTML5. Moreover, users are able to simply drag and drop files into their web browser, which will be further encrypted and stored on a remote server. For the implementation of our web-based encryption tool, which uses the JavaScript Crypto-Framework, we relied on approved and wide-spread cryptographic standards for symmetric and asymmetric encryption, namely AES and RSA.

While the use of our solution provides many benefits to users, in our security analysis we still could identify some limitations. At the moment, it is necessary to trust the application provider that authentic and integer libraries are provided. Additionally, we presented two further scenarios, how this major drawback in our application could be solved. Nevertheless, the best way to improve security will be the inclusion of an implementation of the W3C Cryptography API in standard web browsers.

We successfully demonstrated our solution by implementing a web application, which relies on our JavaScript Crypto-Framework and supports secure storage of encrypted files on a remote server. Future work will include adapting the JavaScript Crypto-Framework according to the W3C Cryptography API, accessing hardware security tokens through JavaScript to improve security in key management, and enabling secure storage at real public cloud providers such as DropBox.

REFERENCES

Barth, A. 2011. The Web Origin Concept. *RFC 6454 (Proposed Standard)*

Diallo, M. et al. 2012. Cloudprotect: Managing data privacy in cloud applications. *IEEE 5th International Conference on Cloud Computing*, Honolulu, HI, pp. 303–310

Dropbox, 2013. Where does dropbox store everyone's data? https://www.dropbox.com/help/7/en.

ECMA International, 2011. *Standard ECMA-262 - ECMAScript Language Specification. 5.1 edition*.

Fielding, R. et al. 1998. *Hypertext Transfer Protocol – HTTP/1.1*. Technical report.

Hickson, I. 2009a. Web storage. Last call WD, *W3C*. http://www.w3.org/TR/2009/WD-webstorage-20091222/.

Hickson, I. 2009b. Web workers. Last call WD, *W3C*. http://www.w3.org/TR/2009/WD-workers-20091222/.

Kaliski, B. 2000. PKCS #5: Password-Based Cryptography Specification Version 2.0. *RFC 2898*

Kandukuri, B. R., Paturi, et al, 2009. Cloud Security Issues. *IEEE 9th International Conference on Services Computing*, Bangalore, pp 517–520.

Kent, S. 1993. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. *RFC 1422*

Leitold, H., et al. 2002. Security Architecture of the Austrian Citizen Card Concept. *18th Annual Computer Security Applications Conferenc*, Las Vegas, pp. 391-400.

Matasano Security, 2010. Javascript cryptography considered harmful. http://www.matasano.com/articles/javascriptcryptography/.

National Institute and Technology, 2001. Advanced encryption standard. *NIST FIPS PUB 197*.

Popovic, K. and Hocenski, Z. 2010. Cloud computing security issues and challenges. *IEEE 33th International Convention (MIPRO)*, pp. 344–349

Ramsdell, B. and Turner, S. 2004. Secure/multipurpose internet mail extensions (s/mime) version 3.1 message specification, *RFC 3851*.

Seiger, R. et al. 2011. Seccsie: A secure cloud storage integrator for Enterprises. *IEEE 13th Conference* on In Commerce and Enterprise Computing (*CEC)*, Luxembourg, pp. 252–255.

Stark, E., et al. 2009. Symmetric cryptography in Javascript. *IEEE 9th Computer Security Applications Conference*, Honolulu, HI, pp. 373–381.

Teufl, P. and Suzic, B. 2012. CCE3 Dokumentation (Version 3.2.0). https://demo.asit.at/resources/buergerkarte/cce2 tool/common/CCE3-Dokumentation-Deutsch.pdf.

W3C, 2011. XMLHttpRequest Level 2: *W3C Working Draft* 16 August 2011. http://www.w3.org/TR/2011/WDXMLHttpRequest2-20110816/.

W3C, 2012. HTML5. A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/.